Animation und Lösung des Rubik-Würfels und des MiniTet

Daniel Beer und Henning Thielemann

Projektarbeit im Sommersemester 1999

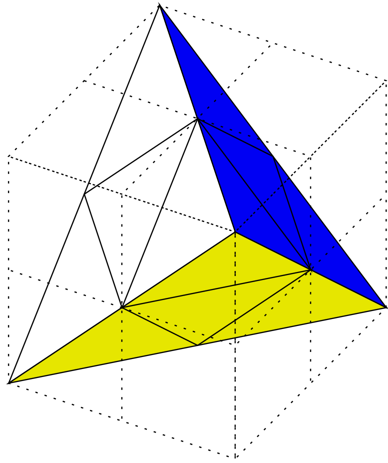
Inhaltsverzeichnis

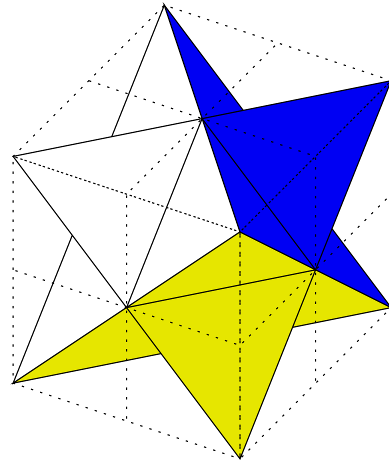
1 Einleitung **3**

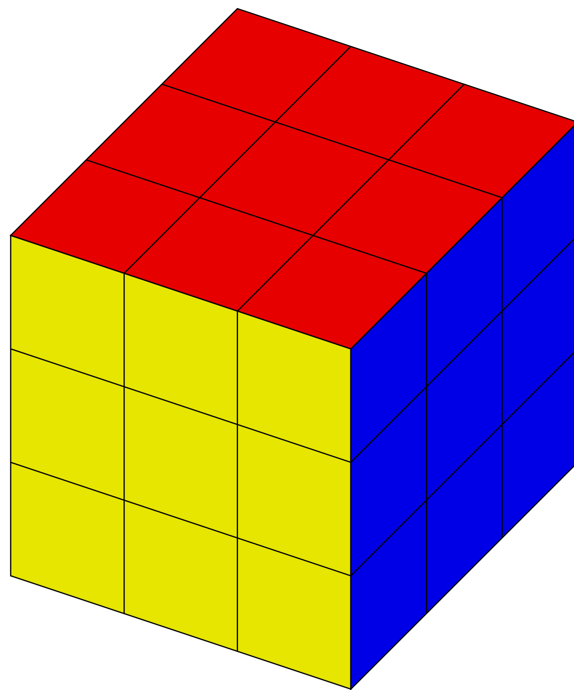
2 Grafische Darstellung **5**
- 2.1 Verwaltung der Daten . . . 5
  - 2.1.1 C++ -Klassen . . . 5
  - 2.1.2 Datenstrukturen für die Puzzles . . . 5
  - 2.1.3 Drehoperationen . . . 6
  - 2.1.4 Verwendung einer Zugliste . . . 6
- 2.2 Darstellung . . . 7
  - 2.2.1 Fensterverwaltung . . . 7
  - 2.2.2 Zeichnen der Puzzles . . . 7
  - 2.2.3 Hierarchische Transformation . . . 8
  - 2.2.4 Bitmapgrafik . . . 9
- 2.3 Bedienung/Steuerung . . . 9
  - 2.3.1 Einleitung . . . 9
  - 2.3.2 Menüs . . . 9
  - 2.3.3 Rotation . . . 9
  - 2.3.4 Ausführen von Zügen . . . 12
  - 2.3.5 Umfärben des Puzzles . . . 13

3 Automatische Lösung **13**
- 3.1 Das magische Tetraeder (Minitet) / Stern . . . 13
  - 3.1.1 Theoretische Grundlagen . . . 13
  - 3.1.2 Testumgebung . . . 13
  - 3.1.3 Datenstrukturen und Operationen . . . 14
  - 3.1.4 Lösungsalgorithmen . . . 14
  - 3.1.5 Schnittstelle zur Grafik . . . 18
  - 3.1.6 Automatische Tests . . . 19
- 3.2 Rubiks Würfel . . . 19
  - 3.2.1 Theoretische Grundlagen . . . 19
  - 3.2.2 Testumgebung . . . 19
  - 3.2.3 Datenstrukturen und Operationen . . . 19
  - 3.2.4 Lösungsalgorithmen . . . 20
  - 3.2.5 Überführung unterschiedlicher Stellungen . . . 23
  - 3.2.6 Automatische Tests . . . 23

4 Literatur **24**







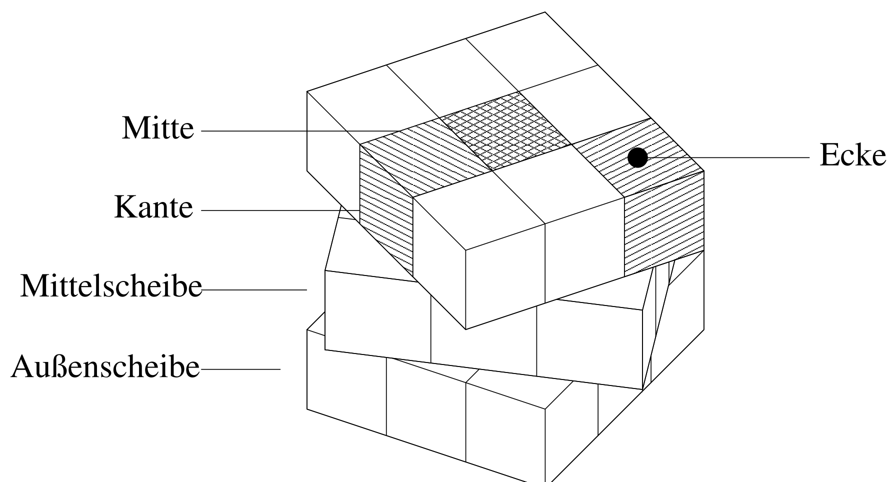


Abbildung 1: Aufbau des Rubik-Würfels

1 Einleitung

Am 30.01.1975 meldete Ernö Rubik, Professor an der Budapester Hochschule für Angewandte Kunst, eine Erfindung zum Patent an, die, um mit den Worten von D.Singmaster zu sprechen, das wahrscheinlich lehrreichste Spielzeug ist, das jemals erfunden wurde. Es handelt sich um den legendären Zauberwürfel, nach seinem Erfinder auch Rubik-Würfel genannt, der inzwischen in über 20 Millionen Exemplaren verkauft wurde.

Gegenstand dieser Arbeit sind die Darstellung und automatische Lösung dieses und zweier weiterer kniffliger räumlicher Puzzles mit Hilfe eines Computers.

Zunächst jedoch soll der Aufbau und die Funktionsweise der 3 Puzzles beschrieben werden.

Rubik-Würfel Im Grundzustand sind alle sechs Seiten einfarbig (im Original: rot, orange, gelb, weiß, blau und grün). Jede dieser Seiten besteht aus 9 Teilfächen (Quadrate); der ganze Würfel scheint aus $3 \times 3 \times 3 = 27$ Teilwürfeln (Segmenten) zu bestehen, von denen allerdings nur 26 zu sehen sind. Jeweils neun Segmente bilden eine Scheibe, die um ihren Mittelpunkt drehbar ist.

Die Segmente werden nach ihrer Lage bzw. nach der Anzahl ihrer farbigen Aufkleber unterschieden. Es gibt also 8 Eckenwürfel, 12 Kantenwürfel und 6 Mittenwürfel, die bei Drehung einer Scheibe wieder in Segmente gleicher Art übergehen. Man wird feststellen daß sich die Mitten wie ein starres Gebilde verhalten, das heißt sie verändern ihre relative Lage zueinander nicht. Kanten und Ecken dagegen gehören mehreren Scheiben gleichzeitig an und wandern somit beim Drehen der selben über die Würfeloberfläche, was den eigentlichen Reiz dieses Puzzles ausmacht. Neben Positionsänderungen der Segmente finden auch Orientierungsänderungen statt. Kanten können gekippt auftreten, Ecken sind möglicherweise gedreht.

Dadurch ist eine fast unausschöpfliche Vielfalt von verschiedenen Stellungen möglich, deren genaue Anzahl später noch berechnet wird.

MiniTet und Stern Diese beiden Puzzles bestehen im Gegensatz zum Rubik-Würfel nur aus 8 Segmenten. Die Drehebenen stehen aber ebenfalls paarweise senkrecht aufeinander. Bei genauer Betrachtung entpuppen sie sich als das reine Eckenproblem des Rubik-Würfels, mit einer wichtigen Einschränkung: vier der acht Ecken sind orientierungsfrei, was zwar die Stellungsvielfalt einschränkt keinesfalls jedoch den Spaß beim Drehen und Lösen.

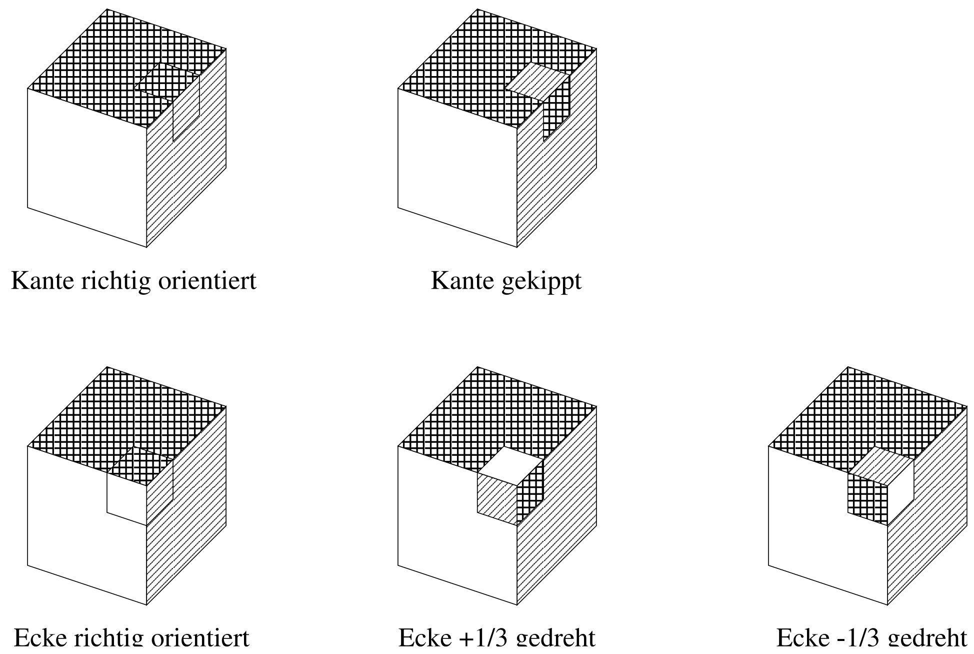


Abbildung 2: Codierung der Orientierung von Würfelteilen

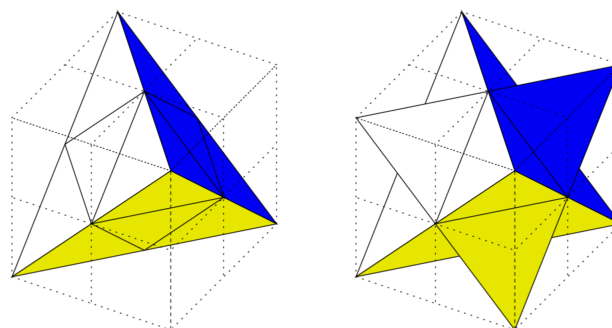


Abbildung 3: Aufbau von Tetraeder und Stern

2 Grafische Darstellung

Die grafische Umsetzung sollte mit Hilfe eines leistungsfähigen Grafiksystems geschehen, welches sowohl das Zeichnen von Primitiven wie Polygonen, Linien etc. beherrschen sollte, sowie in der Lage sein sollte Beleuchtungsberechnungen durchzuführen. Die Suche nach einem solchen Werkzeug endete sehr schnell bei der OpenGL.

Was ist OpenGL, wo kommt sie her und was kann sie?

Bei OpenGL handelt es sich um eine Grafik-Bibliothek (Graphics Library). Diese wurde ursprünglich von der Firma Silicon Graphics entwickelt und nannte sich zunächst nur GL. Diese GL wurde ständig weiterentwickelt und erfreute sich wachsender Beliebtheit. Schließlich entstand ein Nachfolger, der für alle Plattformen offen ist; deshalb OpenGL. Allerdings ist OpenGLTM ein eingetragenes Warenzeichen und damit streng lizensiert. Für alle Benutzer von X11-Systemen ist deshalb die Grafikbibliothek MESA interessant. Diese ist nämlich kostenlos, und der Name wurde vom Autor Brian Paul frei erfunden. Von der Funktionalität her steht MESA der OpenGL inzwischen in nichts nach.

Zusätzlich zum Kern der OpenGL werden noch zwei weitere Bibliotheken mitgeliefert. Es handelt sich um die OpenGL Utility Library (GLU) und das OpenGL Utility Toolkit (GLUT). Während die erste Erweiterung unter anderem die vereinfachte Handhabung der geometrischen Transformationen sowie die Erstellung von Kugeln, Zylindern, u. ä. unterstützt, ermöglicht GLUT die Einbindung der Applikation in ein Fenstersystem.

Zum Schluß noch ein kurzer Überblick über die wichtigsten Fähigkeiten der OpenGL:

- 3D-Grafik, dazu gehören schattierte Punkte, Linien, Polygone Stacks für Attribute und Matrizen Ausblenden verdeckter Objekte (Depth Buffering) Arbeit mit Displaylisten Glätten von Linien und Polygonkanten (Antialiasing)
- Materialeigenschaften, Beleuchtung, Texturierung, atmosphärische Effekte
- verschiedene Puffer für Spezialeffekte

u. v. m.

2.1 Verwaltung der Daten

2.1.1 C++ -Klassen

Das Programm wurde in C++ geschrieben und, soweit dies möglich war, objektorientiert gestaltet. Diese Ausdrucksweise läßt erkennen, daß es diesbezüglich Einschränkungen gab. Zur Ereignisüberwachung werden von der Utility Bibliothek von OpenGL sogenannte Callback-Hooks benutzt (dazu später mehr). Diese rufen im Falle eines Ereignisses eine Funktion auf, die allerdings global sichtbar sein muß. Es kann also keine Methode einer Klasse benutzt werden, was das Integrieren von Fenstern in eine Klasse unmöglich macht.

Fensterunabhängige Daten dagegen werden konsequent in Klassen verwaltet. Dabei erhielt der Rubik-Würfel eine eigene Klasse, für Stern und MiniTet hingegen wurde eine gemeinsame Klasse implementiert, da sich beide Puzzles nur minimal im Aussehen unterscheiden. Die Funktionsweise ist völlig identisch. Somit reicht bei der Objekterzeugung nur ein Flag zur Unterscheidung.

Weiterhin wurde eine Klasse zur Verwaltung von Schaltflächen (Gadgets) eingeführt. Sie enthält Informationen über Aussehen, Größe und Lage der Bereiche in denen per Mausknopf eine Aktion ausgelöst werden soll. Die zugehörigen Methoden dienen der Darstellung bzw. dem Test ob sich der Mauszeiger innerhalb eines der oben genannten Bereiche befindet.

Schließlich wurde noch eine Klasse implementiert, mit deren Hilfe einfache geometrische Objekte wie Polygone und Linien in einen Puffer gezeichnet werden können.

2.1.2 Datenstrukturen für die Puzzles

Bei der Speicherung des Aussehens der einzelnen Puzzles wurde darauf geachtet, daß die darauf anzuwendenden Operationen, in erster Linie die Darstellung, relativ einfach zu implementieren sind. Somit ist es nicht verwunderlich, daß die Daten zur Lösungsfindung anders abgelegt wurden.

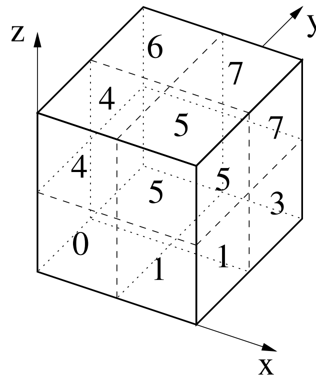


Abbildung 4: Organisation des $2 \times 2 \times 2$-Würfels

Stern Jede Ecke besteht aus einem Feld mit 3 Einträgen, in denen die Farbwerte (0 bis 2) der drei Aufkleber auf der Ecke gespeichert sind. Die Einträge entsprechen den Farbwerten entgegen dem Uhrzeigersinn. Der erste und dritte Eintrag enthält die Farben der oberen bzw. unteren beiden Aufkleber einer Ecke, je nach dem ob sich diese in der oberen bzw. unteren Hälfte des Sterns befindet. Der zweite Eintrag verweist demnach auf die zur Mitte zeigenden Flächen. Die Ecken selbst sind wiederum in einem Feld abgelegt,

MiniTet Die Datenstruktur dieses Puzzles ist völlig identisch aufgebaut wie die des Sterns. Auch hier werden für jedes Segment drei Farbwerte abgelegt, obwohl für die einfarbigen Segmente ein Wert gereicht hätte. Auf diese Weise bleiben die Datenstrukturen einfach und vor allem kompatibel.

Rubik-Würfel Im Gegensatz zum MiniTet wird der Würfel nicht segmentweise sondern flächenweise abgespeichert. Jede der sechs Außenflächen besteht bekanntlich aus 9 Aufklebern. Es wird also ein 2-dim. Feld mit der Ausdehnung 6x9 benötigt. Die genaue Indizierung ist auch hier der Skizze zu entnehmen.

2.1.3 Drehoperationen

Drehen bedeutet prinzipiell, daß beliebig viele Scheiben einer Drehebene in eine bestimmte Richtung um einen bestimmten Winkel gedreht werden. Der Winkel ist in diesem Fall immer ein Vielfaches von $90°$. Die Richtung wird nicht explizit angegeben, sondern ist das Vorzeichen des Winkels. Da eine $360°$-Drehung nichts bewirkt und eine Drehung um $270°$ einer Drehung um $-90°$ entspricht, sind nur folgende Drehwinkel zugelassen: $-90°$, $90°$ und $180°$. Diese werden kodiert als -1, 1 und 2. Die Drehebene wird spezifiziert durch die Drehachse, welche entweder in Richtung der x-, y- oder z-Achse zeigt. Als Kodierung wird hier 0, 1 und 2 verwendet. Schließlich muß noch angegeben werden, welche Scheiben gedreht werden sollen. Der Würfel besitzt in jeder Drehebene 3, das MiniTet nur 2 Scheiben. Auch hier wird wieder eine Kodierung eingeführt. Die Scheiben erhalten Zweierpotenzen - beginnend mit 20 - als Identifikation. Sollen mehrere Scheiben bewegt werden, müssen diese Zahlen addiert werden. Ein Wert von $7 = 111_2$ (Binärdarstellung) beispielsweise bedeutet, daß sich alle drei Scheiben, also der ganze Würfel, drehen.

2.1.4 Verwendung einer Zugliste

Um Züge zurücknehmen zu können, ist es erforderlich, eine Zugliste für bereits ausgeführte Züge zu verwalten (im Programmtext History-Liste genannt). Diese wird intern als doppelt verkettete Liste realisiert. Die Listenelemente enthalten den zu diesem Zeitpunkt gerade ausgeführten Zug, in der im letzten Abschnitt beschriebenen Kodierung. Das Zurücknehmen eines Zuges bedeutet also die Werte des aktuellen Listenelements mit negiertem Winkel an die Drehroutine zu übergeben und anschließend den Vorgänger zur aktuellen Listenposition zu machen. Im Gegensatz dazu müssen beim erneuten Ausführen eines gegebenenfalls vorher zurückgenommenen Zuges zuerst der Nachfolger des aktuellen Listenelements bestimmt und anschließend seine Daten an die Drehroutine übergeben werden.

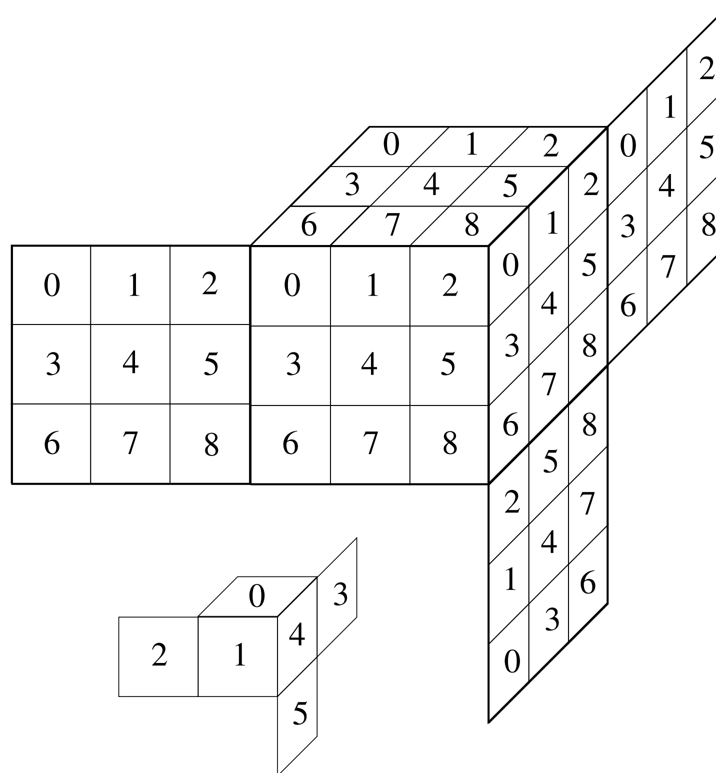


Abbildung 5: Organisation des $3 \times 3 \times 3$-Würfels

Werden zusätzliche Züge in die Zugliste eingefügt, zum Beispiel weil der Benutzer einen Zug ausführt oder das Puzzle ordnen läßt, geschieht dies grundsätzlich nach der aktuellen Listenposition. Alle Züge, die bezüglich dieser Position in der Zukunft liegen, werden dabei gelöscht!

Doch die Zugliste kann auch noch anders eingesetzt werden. So werden vom Computer erzeugte Züge (Zufallszüge oder Züge zum Lösen des Puzzles) nicht direkt ausgeführt, sondern zuerst an die Liste angehängt. Dem Benutzer wird damit freigestellt, ob die Züge tatsächlich angewendet werden.

2.2 Darstellung

2.2.1 Fensterverwaltung

Eine übersichtliche Bedienoberfläche verlangt die Benutzung von Fenstern. Diesbezüglich läßt OpenGL fast keine Wünsche offen. Es lassen sich beliebig viele Fenster öffnen, alle mit einem eigenen Kontext ausgestattet, d.h. Benutzereingaben und andere Ereignisse können getrennt nach Fenstern abgefangen und bearbeitet werden. Solche Ereignisse sind im wesentlichen Tastatur- und Mauseingaben, das Vergrößern und Anzeigen/Verbergen von Fenstern, sowie das Ablaufen einer festgelegten Zeitspanne. Um auf jede Situation reagieren zu können, werden so genannte Callback-Funktionen (Hooks) initialisiert. Ist die Hauptschleife einmal gestartet, werden bei eintreffenden Ereignissen automatisch die entsprechenden Funktionen ausgeführt. Die einzige Ausnahme stellt der Idle-Hook dar, der nicht auf ein spezielles Ereignis wartet, sondern zu jedem möglichen Zeitpunkt die ihm zugeteilte Funktion ausführt. Außerdem ist dieser Hook nicht fensterabhängig, d.h. es wird für das gesamte Programm nur ein Idle-Hook benötigt.

2.2.2 Zeichnen der Puzzles

Die Oberfläche der Puzzles besteht aus gleichseitigen Dreiecken bzw. Quadraten, wobei die Kanten und Ecken abgerundet sind. OpenGL ist zwar in der Lage Polygone wie Dreiecke und Vierecke (welche intern

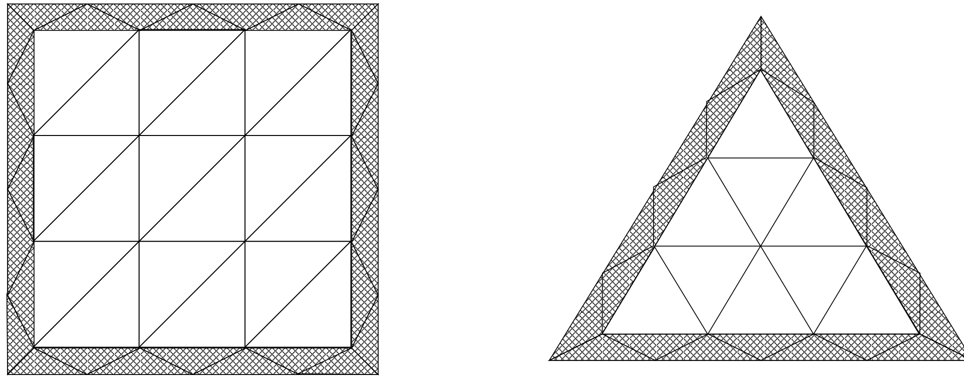


Abbildung 6: Triangulierung der Würfelseitenflächen mit Aufklebern

in zwei Dreiecke zerlegt werden) zu zeichnen, jedoch nur in einer bestimmten Farbe. Bekanntlich hat jedes Farbsegment der Puzzles einen schwarzen (und somit andersfarbigen) Rand. Es ist also erforderlich, die Oberfläche der Puzzles weiter in Polygone zu zerlegen. Doch es gibt noch mehr zu beachten. Lichtreflexionen, die für ein realistisches Aussehen unabdingbar sind, werden nur an den Eckpunkten der Polygone (also Dreiecke) berechnet. Der Farbwert an den übrigen Punkten der Dreiecksfläche wird linear interpoliert. Das ist eine gewisse Einschränkung, erhöht aber drastisch die Darstellungsgeschwindigkeit. In bestimmten Fällen treten dadurch starke Verfälschungen auf, nämlich dann, wenn ein Schlaglicht innerhalb der Dreicksfläche erscheinen müßte. Dieses bleibt aber auf Grund der Interpolation unsichtbar. Einzige Abhilfe ist eine feinere Unterteilung der Polygone, in der Hoffnung, daß Schlaglichter immer mindestens auf eine Ecke treffen und somit sichtbar werden. Die Feinheit der Unterteilung der Segmente wird im Programm vom Parameter PRECISION gesteuert; ein Wert von 3 beispielsweise liefert folgende Aufteilung:

Auf diese Weise erhält man ein recht realistisches Aussehen, mit einer Ausnahme: es fehlen noch die abgeschrägten Ecken und Kanten. Auch hier wäre es möglich ein feines Polygonnetz zu verwenden, welches bogenförmig die Kanten überzieht. Damit würde sich aber wieder der Rechenaufwand erhöhen. Als Abhilfe wird ein kleiner Trick angewendet. Statt auftreffendes Licht immer senkrecht zur Oberfläche reflektieren zu lassen, werden die Normalenvektoren der Eckpunkte der Polygone absichtlich verfälscht. Die Lichreflexion wird damit so verändert, als ob die Kanten tatsächlich abgeschrägt wären. Dieses Vorgehen wird auch als „Bump mapping“ bezeichnet.

Schließlich muß noch bedacht werden, daß beim Ausführen von Zügen zwischenzeitlich das Innere der Puzzles sichtbar wird. Damit die Puzzles nicht hohl wirken, werden zusätzliche schwarze Flächen an den Drehebenen eingefügt.

Wie sicherlich deutlich wurde, bestehen die einzelnen Farbsegmente der Puzzles aus einer Vielzahl von Polygonen. Diese müssen bei jeder Lageveränderung des Puzzles in einer langen Sequenz für jedes Segment neu erzeugt werden. Um den Programmcode nicht explodieren zu lassen, bietet OpenGL die Möglichkeit, eine Folge von Zeichenoperationen in einer sogenannten Displayliste abzuspeichern. Dabei wird zusätzlich versucht eventuell auftretende Transformationen intern zu vereinfachen, was eine Rechenzeitverkürzung zur Folge hat.

2.2.3 Hierarchische Transformation

Das Zeichnen der Polygone untergliedert sich in folgende Schritte. Zunächst muß eine Transformation des ursprünglichen Koordinatensystems durchgeführt werden. Diese umfaßt zum einen die Verdrehung des Puzzles um seine Längsachse sowie die Kippung nach vorn oder hinten und zum anderen die Verschiebung zum entsprechenden Segment sowie die korrekte Plazierung innerhalb diesem. Diese Kette von einzelnen Transformationen muß nun für jedes Primitiv durchgeführt werden. Wie man sich leicht vorstellen kann, ist diese Vorgehensweise ziemlich rechenintensiv, zumal nicht nur Matrixmultiplikationen, sondern auch trigonometrische Berechnungen durchgeführt werden müssen.

Auch hier bietet OpenGL Abhilfe. Es ist möglich, bereits durchgeführte Transformationen in Form von

Matrizen auf einem Stack abzulegen. So würde man zunächst die Verdrehung und Kippung durchführen, diese Matrix sichern, und erst jetzt die nächste Transformation berechnen. Verfährt man so weiter, spart man eine Menge Rechenzeit ein, denn einmal berechnete Matrizen müssen nur noch vom Stack geholt werden. Außerdem wird das Programm um einiges übersichtlicher.

Die Folge von Transformationen und Stackoperationen läßt sich zur Veranschaulichung als Baum darstellen.

Jetzt gilt es noch, die Animationsphasen der Puzzles während der Zugausführug zu betrachten. Da in diesem Fall einige Teile zum restlichen Puzzle verdreht sind, muß die Darstellung schichtweise geschehen (statt sonst segmentweise, Seite für Seite), wobei zum Zeichnen der verdrehten Schichten eine zusätzliche Transformation ausgeführt wird.

Um auch hier einen realistischen Bewegungsablauf zu erhalten, folgen die Drehwinkel der Animation einer kubischen Gleichung. Es handelt sich also um eine ungleichmäßig beschleunigte Bewegung.

2.2.4 Bitmapgrafik

Während bisher nur über von OpenGL gerenderte Grafik gesprochen wurde, jetzt noch einige Bemerkungen zu Bildgrafiken, die auch im Programm zur Gestaltung der Bedienoberfläche benutzt wurden.

Grafiken werden vom Dateisystem in den Hauptspeicher geladen und müssen in einem bestimmten Format abgelegt sein, welches OpenGL verarbeiten kann. Es handelt sich leider nicht um ein gebräuchliches Grafikformat, sondern um ein Rohformat ohne jegliche Informationen über Größe und Farbtiefe. Das Bild wird zeilenweise von unten nach oben (da bei OpenGL der Koordinatenursprung unten links liegt!) abgelegt. Jeder Punkt des Bildes wird durch ein Zahlentripel repräsentiert, welches die Rot-, Grün- und Blauanteile des Pixels widerspiegelt. Ob es sich um 1-, 2- oder 4-Byte-Werte handelt ist unwichtig, solange man beim Einladen die korrekte Größe angibt. Einmal im Speicher kann die Grafik mit einem einfachen Befehl an eine beliebige Position im Fenster kopiert werden. Leider läuft die Grafik zuvor durch die komplette Renderpipeline, so daß Skalierungen, Drehungen und andere Transformationen sowie Clipoperationen und Farbmischungen wirksam werden können (auch wenn sie nicht benötigt werden). Das verlangsamt die Darstellung merklich. Allerdings kümmert sich OpenGL selbst darum, daß die Farben korrekt aussehen, eventuell wird auch gerastert (Dithering), was gute Ergebnisse liefert.

2.3 Bedienung/Steuerung

2.3.1 Einleitung

Ein gutes Programm zeichnet sich durch eine einfache und intuitive Bedienung aus. Welches Eingabegerät eignet sich da besser als eine Maus? Daher wurde viel Arbeit investiert, um eine leicht verständliche Maussteuerung zu implementieren. Natürlich wurde auch auf diejenigen Nutzer Rücksicht genommen, die eine Tastatur der Maus vorziehen, indem wichtige Funktionen auch mittels Tastatur ausführbar sind.

2.3.2 Menüs

Mit den Funktionen der GLUT-Bibliothek kann man auf einfache Weise beliebig tief geschachtelte Menüs erzeugen. Jeder Menüpunkt erhält eine Nummer mit der er eindeutig identifiziert werden kann. Desweiteren muß das Menü mit einer beliebigen Maustaste verknüpft werden, in diesem Fall, wie allgemein üblich, die rechte Taste. Leider ist es nicht möglich Balken zur Trennung der Menüeinträge einzufügen oder Einträge mit Schaltern zu versehen. Allerdings kann man zur Laufzeit Menüeinträge löschen oder hinzufügen.

Um Menüauswahlen abfangen zu können, muß ein entsprechender Callback-Hook deklariert werden. Er liefert die Nummer des Menüpunkts, die bei der Initialisierung angegeben wurde.

2.3.3 Rotation

Nicht alle Seiten des Puzzles sind gleichzeitig sichtbar. Durch Rotation des Puzzles kann aber ein kompletter Überblick über das Aussehen des Puzzles vermittelt werden.

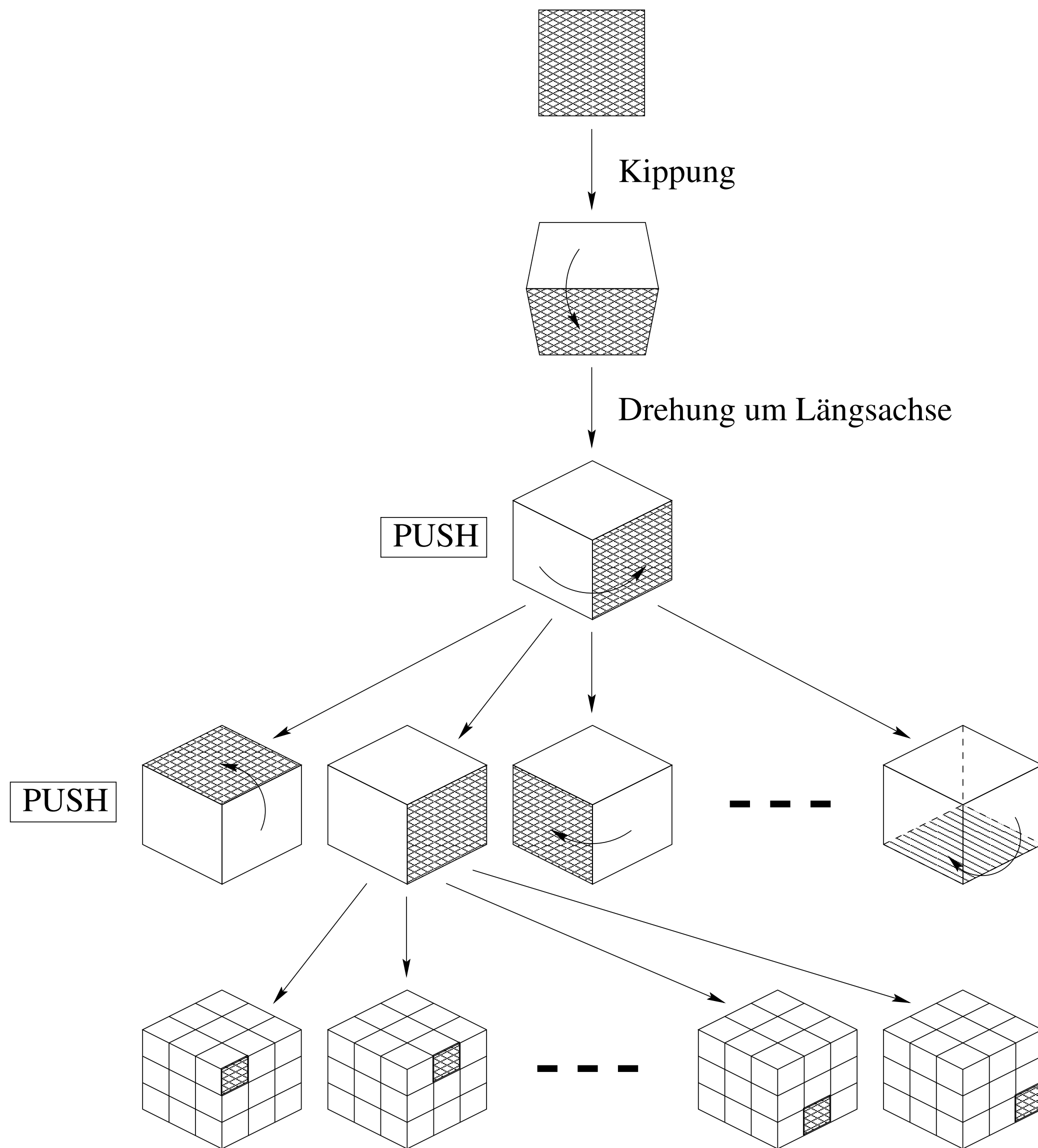


Abbildung 7: Schritte zur 3D-Darstellung

Ansatz:
$a(t) = ct + a_0$ (Beschleunigung)
$v(t) = \frac{1}{2}ct^2 + a_0 t + v_0$ (Geschwindigkeit)
$s(t) = \frac{1}{6}ct^3 + \frac{1}{2}a_0 t^2 + v_0 t + s_0$ (Weg)

folgendes ist bekannt:
$s(0) = 0,\ s(1) = 90$ (Rotation um 90 Grad)
$v(0) = 0,\ v(1) = 0$ (Anfangs – und Endgeschwindigkeit sind Null)

$\Rightarrow\ 0 = v(0) = v_0$
$0 = v(1) = \frac{1}{2}c + a_0$ (1)
$0 = s(0) = s_0$
$90 = s(1) = \frac{1}{6}c + \frac{1}{2}a_0$ (2)
(1), (2) $\Rightarrow\ a_0 = 540,\ c = -1080$

einsetzen:
$s(t) = -180t^3 + 270t^2 = 90t^2(3 - 2t)$

diskretisieren:
$subst:\ t = \frac{j}{step},\ step = Anzahl\ der\ Schritte$
$\Rightarrow s(j) = 90\frac{j^2}{step^2}(3 - 2\frac{j}{step}) = \frac{90j^2(3step - 2j)}{step^3},\ 0 \leq j \leq step$

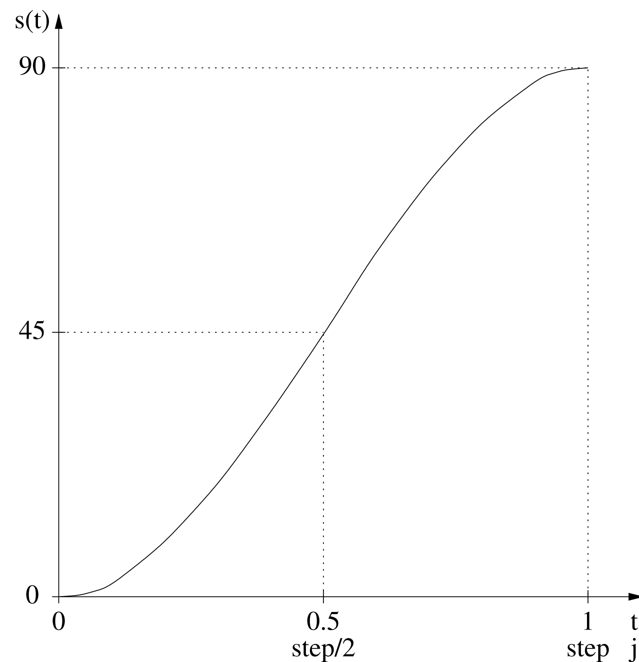


Abbildung 8: Beschleunigte Bewegung

Taste	Bedeutung
l	links (left)
i	mitte
r	rechts (right)

Taste	Bedeutung
t	oben (top)
o	mitte
d	unten (down)

Taste	Bedeutung
f	vorne (front)
p	mitte
b	hinten (back)

Tabelle 1: Tastenbelegung

Die Rotationsachse verläuft durch die Deck- und Grundfläche des Puzzles, und kann um maximal 90° nach vorn oder hinten gekippt werden. Man erhält ein Modell ähnlich eines Globus. Diese Einschränkung stellt sicher, daß man nicht die Orientierung verliert.

Es ist wichtig zu erwähnen, daß sich das Puzzle eigenständig mit konstanter aber regelbarer Geschwindigkeit um die Längsachse dreht, während für die Kippung der Winkel einstellbar ist.

Die Rotation kann sowohl mit der Tastatur (Cursortasten) als auch mit der Maus gesteuert werden. Sobald die mittlere Maustaste gedrückt wird, folgt das Puzzle den Bewegungen der Maus. Nachdem man die Taste wieder losläßt, bleiben die aktuelle Rotationsgeschwindigkeit und der Kippwinkel erhalten.

2.3.4 Ausführen von Zügen

Züge können auf drei verschiedene Arten angewiesen werden. Die einfachste Methode ist die Benutzung der Tastatur. Unter Zuhilfenahme von Shift- und Controltaste lassen sich fast alle Züge ausführen. Hier eine Übersicht über die Tastenbelegungen:

Die zweite, zugleich die eher gewöhnungsbedürftige, Methode ist die Benutzung von Schaltflächen. In einem separaten Fenster sind alle möglichen Züge als Knöpfe mit Pfeilen, die Drehebene und -richtung andeuten, in einem Raster angeordnet. Die Pfeile entsprechen den Drehungen, wenn man von Süd nach Nord auf das Puzzle schaut. Ein Klick mit der linken Maustaste auf einen Knopf aktiviert sofort die entsprechende Aktion.

Die letzte Methode erlaubt einen sehr intuitiven Umgang mit dem Puzzle. Um einen Zug auszuführen, genügt es mit der Maus direkt „in das Puzzle hineinzugreifen“ und die ausgewählte Scheibe zu drehen. Diese Aktion ist allerdings nur beim Rubik-Würfel implementiert.

Der Rubik-Würfel besteht aus 9 Scheiben. Es mußte sichergestellt werden, daß die Auswahl einer Scheibe mit der Maus eindeutig ist. Damit kamen die Farbsegmente als Auswahlelemente nicht in Frage, da sie jeweils zu zwei Scheiben gehören. Einzig die Würfelkanten gestatten eine eindeutige Identifikation, denn keine Kante gehört zu mehreren Scheiben. Zudem sind von jeder Scheibe immer mindestens 2 Kanten zu sehen, so daß in jeder Lage des Würfels alle Züge ausführbar sind. Einzige Ausnahme ist die Ansicht senkrecht auf eine der sechs Flächen. Hier sind Drehungen, die in der Bildebene liegen nicht möglich.

Man erkennt, daß diese Lösung für MiniTet/Stern wenig geeignet ist, denn diese besitzen keine Entsprechungen für die Kanten des 23-Würfels, in den sie einbeschrieben sind.

Ein schwieriges Problem bei der Umsetzung der Idee war die Assoziation von 2-dimensionalen Mauskoordinaten mit 3-dimensionalen Raumkoordinaten. Um genauer zu sein: Wie erhält man aus gegebener Mausposition die Scheibe, die gedreht werden soll? Es wäre sicher möglich mittels geometrischer Berechnungen dieses Problem zu lösen. Doch schien dies recht kompliziert zu sein. Auch in Hinsicht auf ähnliche Probleme, die beim Umfärben der Farbsegmente auftraten (s. nächster Abschnitt), fiel die Entscheidung auf eine alternative Bewältigung dieser Aufgabe. Es wurde eine Datenstruktur implementiert, die es erlaubt in einen Puffer, der sich an die Größe des Ausgabefensters anpaßt, ausgefüllte Polygone und Linien beliebiger Dicke in einer bestimmten Farbe zu zeichnen. Die Tiefensortierung wurde mittels Z-Puffer-Methode gelöst.

Mittels dieser Datenstruktur war es nun möglich ein Abbild des Rubik-Würfels in vereinfachter Form zu erstellen. Der Würfelkörper wurde durch 6 Quadrate in der Hintergrundfarbe nachgebildet. Damit ist er zwar unsichtbar, doch Bildteile die sich räumlich dahinter befinden, werden davon verdeckt. An jede Kante wurde eine dicke Linie mit aufsteigender Farbnummer gezeichnet. Wie bereits erwähnt, sind jetzt nur die im Vordergrund liegenden Kanten sichtbar.

Zum Herausfinden der ausgewählten Kante muß nur noch der Farbwert an der Mausposition ausgelesen werden. Ist der Punkt in der Hintergrundfarbe gezeichnet, befindet sich der Mauszeiger nicht über einer

Kante.

2.3.5 Umfärben des Puzzles

Damit das Programm auch praktisch sinnvoll genutzt werden kann, ist es nicht ausreichend nur Züge auszuführen zu können. Für den Fall, daß man eine konkrete Start- oder Zielstellung vorgeben möchte, ist es notwendig, das Puzzle entsprechend färben zu können.

Das geschieht wieder am einfachsten mit der Maus. In einem zusätzlichen Fenster wählt man eine der zur Verfügung stehenden Farben aus und überträgt sie mittels Mausklick auf ein Farbsegment seiner Wahl.

Auch hier wurde die im vorigen Abschnitt beschriebene Datenstruktur erfolgreich eingesetzt. Während vorher die Kanten hervorgehoben wurden, werden jetzt die einzelnen Farbsegmente mit verschiedenen Farben (54 beim Rubik-Würfel, 24 beim Stern und MiniTet) markiert. Auf diese Weise kann auch bei relativ bizarren Formen des Puzzles, wie sie beim MiniTet oft auftreten, sehr einfach das unter dem Mauszeiger liegende Farbsegment bestimmt werden.

3 Automatische Lösung

3.1 Das magische Tetraeder (Minitet) / Stern

3.1.1 Theoretische Grundlagen

Das magische Tetraeder/Stern entspricht dem $2 \times 2 \times 2$ Würfel. Es besteht also nur aus Eckenstücken, wobei beim Stern alle Teile als kleine Tetraeder (Spitzenteile) ausgeprägt sind, beim MiniTet hingegen nur vier Teile tetraederförmig sind und die anderen vier nur als Dreiecke (flache Teile) wahrnehmbar sind. Den flachen Teilen sieht man ihre Orientierung nicht an, wir wollen sie deswegen als orientierungsfrei betrachten. Unter dieser Voraussetzung gibt es 136080 verschiedene Stellungen des MiniTets: Wir halten eine orientierbare Ecke fest, dann können die restlichen Ecken auf den verbleibenden sieben Plätzen permutieren (7!) und drei Ecken können verschieden orientiert sein (33). Da im ersten Zug neun verschiedene Zugmöglichkeiten bestehen (90°, 180° oder 270° in jeder der drei Dimensionen) und in jedem weiteren sechs Möglichkeiten (aufeinanderfolgende Züge brauchen nicht in der gleichen Dimension stattzufinden), braucht man wegen $9 \cdot 65 = 69984 < 136080$ mindestens 7 Züge, um alle Stellungen zu erreichen. Durch Überprüfen aller möglichen Stellungen ermittelt man ein Maximum von 9 Drehungen (Winkel der Drehungen beliebig), um eine beliebige Stellung in die Grundstellung zu überführen.

Aufgrund der unterschiedlichen Art der Teile, kann man das MiniTet durch Drehen in andere Formen bringen, es gibt davon bis auf vollständige Drehungen sieben verschiedene.

3.1.2 Testumgebung

Das Entwickeln und Testen der Lösungsalgorithmen soll unabhängig von der grafischen Ausgabe möglich sein. Zum einen müssen Stellungen zum Testen eingegeben und zum anderen müssen die Ergebnisstellungen ausgegeben werden können.

Stellungen können direkt im Programmcode als konstante Felder abgelegt werden. Zur Vereinfachung dürfen dies nur Stellungen in Tetraederform sein. Ein derartiges Feld besteht aus zwei Dimensionen, die erste Dimension adressiert die Seitenfläche (bezogen auf die Tetraedergestalt), die zweite Dimension adressiert die Position des Farbblättchens.

Mit Hilfe eines vorgefertigten Feldes, das jeder Eckenposition die Positionen der drei Farbblättchen in der Tetraederseiten-Beschreibung zuordnet, und einem Feld, das für alle Eckenteile die verwendeten Farbblättchen verzeichnet, erstellt eine Routine aus der Seitenflächenbeschreibung die Datenstruktur, die für die Lösungsalgorithmen benötigt wird.

Für die Kontrollausgabe wird sowohl die interne Datenstruktur verwendet als auch die Seitenflächendarstellung aufbereitet.

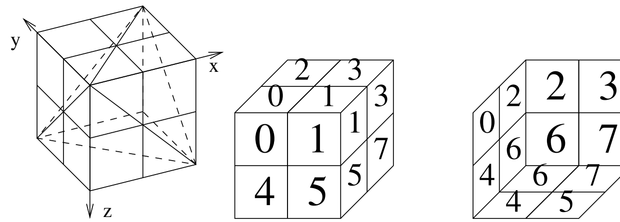


Abbildung 9: Interpretation des Tetraeders als $2 \times 2 \times 2$-Würfel

3.1.3 Datenstrukturen und Operationen

Für gewöhnlich richtet sich die Wahl einer Datenstruktur zuerst nach der Art der Daten, die abgespeichert werden sollen und dann danach, welche Operationen daran ausgeführt werden sollen. Bei der Datenstruktur für das MiniTet benötigen wir Tests auf gültige Stellungen, Drehungen einzelner Scheiben, das Vergleichen mit anderen Stellungen, das einfache Erkennen bestimmter Situationen, z.B. ob sich das Puzzle in Tetraederform befindet. Dabei ist es wichtig, Symmetrien am MiniTet auszunutzen. So ist zum Beispiel eine Scheibendrehung in jeder der drei Dimensionen möglich und es sollte eine einzige Routine reichen, um alle Drehungen in beliebigen Scheiben um eine beliebige Anzahl Vierteldrehungen durchzuführen.

Die Darstellung des MiniTets über seine Seitenansichten ist zum Beispiel für das Lösen ungeeignet, abgesehen davon, daß sie nicht alle Stellungen beschreiben kann, kann man an ihr auch schwer Drehungen durchführen oder Eckenorientierungen erkennen.

Wir wollen uns die Analogie zum $2 \times 2 \times 2$-Würfel zu nutze machen und die Eckenbeschreibungen in einem $2 \times 2 \times 2$-Feld unterbringen. In der Praxis hat sich gezeigt, daß ein linearisiertes Feld einfacher zu handhaben ist, man kann dann zum Beispiel mit einer einfachen Schleife eine Operation auf alle Feldelemente anwenden. Über die Binärzahlendarstellung ist man immer noch in der Lage, recht unkompliziert auf die Elemente zuzugreifen. Element $5 = 10^{12}$ beinhaltet zum Beispiel das Eckenteil an der Position $z = 1, y = 0, x = 1$.

Wie kodiert man nun die Eckenteile? Die naheliegendste Möglichkeit wäre wohl, Kodierungen für die 3 Farbblättchen jeder Ecke abzulegen. Damit kann aber die Orientierung eines Teil nicht so einfach ablesen. Besser ist es deswegen, alle Eckenteile durchzunumerieren - am einfachsten entsprechend ihrer Position im Tetraeder-Grundzustand - und diese Nummern nebst einer Orientierungskodierung zu speichern. Dieses Verfahren hat darüberhinaus den Vorteil, daß man nur reale Eckenteile kodieren kann.

Bleibt die Frage nach der Kodierung der Orientierung. Ist eine Ecke an ihrer Grundposition, bereitet uns die Definition der Orientierung keine Schwierigkeiten. Die normale Orientierung bezeichnen wir mit 0. Drehen wir die Ecke rechts herum, nennen wir die Orientierung $+1$, links herum nennen wir -1. Wie kodieren wir nun die Orientierung eines Teiles, das nicht an seiner Grundposition steht? Wir markieren jedes Teil an einer Seite und betrachten bei einer beliebigen Stelle die Abweichung von markierter Seite der Ecke zu der Seite die markiert wäre, stände dort die Ecke, die dort in der Grundstellung hingehört, als Orientierung. Bleibt noch zu klären, welche Seite eines jeden Teils zu markieren ist. Leider ist es nicht möglich, die Markierungen so zu gestalten, daß die Drehroutine für alle Scheiben gleich ist. Wir versuchen wenigstens ein Muster zu finden, so daß sich die Routine mit wenigen Parametern an die jeweilige Dimension anpassen kann.

Eine solche Musterung zeigt die Abbildung. Man erkennt, daß Rotationen in der oberen Ebene die Orientierungen nicht verändern, da eine markierte Seite immer wieder in eine markierte Seite übergeht. In den anderen Richtungen muß dagegen zu der Orientierung immer im Wechsel eins addiert und eins subtrahiert werden.

3.1.4 Lösungsalgorithmen

Mit dem Lösen eines Puzzles verbindet man intuitiv die Suche nach einer Zugfolge, die von einer vorgebenen Stellung zu einer anderen Stellung führt. Es sollen hier die zwei Strategien vorgestellt werden, die implementiert wurden.

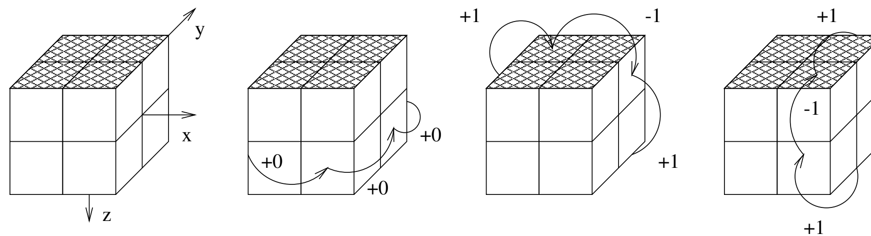


Abbildung 10: Codierung der Orientierung von Tetraeder- und Sternteilen

Vollständige Suche Das Ausprobieren aller denkbarer Zugfolgen ist die einfachste Methode, mit der man zu dem die bewiesenermaßen kürzeste Zugfolge findet. Mit einem gewichtigen Nachteil: Die Zahl der zu testenden Zugfolgen wächst exponentiell mit der Zugfolgenlänge. Trotzdem gelingt es mit der aktuellen Rechentechnik zumindest für das kleine MiniTet alle Zugfolgen in akzeptabler Zeit zu testen. Der gerichtete Graph in dem die Stellungen durch Knoten und die Züge durch Kanten dargestellt sind, besitzt 136080 Knoten und $136080 \cdot 9 = 1224720$ Kanten. Bei maschinennaher Programmierung könnte man mit einem Bitfeld (17010 Bytes), das die bereits erreichten Stellungen registriert, und einem Feld aus 136080 Bytes, das vermerkt mit welchem Zug man zu der zur Adresse gehörigen Stellung gekommen ist, eine Breitensuche sehr schnell realisieren.

Muß man für die Breitensuche allerdings für jede neu erreichte Stellung Speicher reservieren und einen Suchbaum für alle schon erreichten Stellungen verwalten, wird die Breitensuche so langsam, daß man besser auf eine Tiefensuche zurückgreift. Um auch hier die kürzeste Lösung zu erhalten, werden zuerst alle Zugfolgen der Länge 0, dann alle der Länge 1, usw. getestet. Der Nachteil ist, daß alle Sequenzen eines Durchgangs im nächsten wieder komplett durchlaufen werden. Wenn man sich vor Augen führt, daß die Erweiterung des Suchbereiches um einen Zug den Zugfolgenumfang bereits versechsfacht, sieht man leicht ein, daß für die Dauer des gesamten Algorithmus der letzte Schritt ausschlaggebend ist. Auch darüberhinaus erledigt der Tiefensuchealgorithmus noch mehr Arbeit als nötig, aber in der Praxis hat sich gezeigt, daß er der (Hochsprachen-)Breitensuche überlegen ist.

Intuitive Lösung So reizvoll die kürzeste Lösung ist, so wenig lehrreich ist sie auch, denn ein Konzept läßt sich in der Computerstrategie nicht erkennen. Sinnvoll ist deswegen ein weiterer Lösungsweg, der dem menschlichen Denken entgegenkommt. Dabei wird die Lösung des Puzzles in Etappen unterteilt, deren Zwischenegebnisse man leicht visuell erfassen kann. Je nachdem, ob man das MiniTet als Tetraeder betrachtet, oder in ihm die Vereinfachung des $3 \times 3 \times 3$-Würfels erkennt, gibt es verschiedene Etappenaufteilungen. Im letzteren Fall übernimmt man eine Strategie des Sortierens der Ecken am Rubik-Würfel. Zum Beispiel ordnet man erst eine Scheibe, dann plaziert man in der anderen Scheibe alle Ecken, dann orientiert man diese. Problematisch ist nur, daß man notfalls auch eine überflüssig erscheinende Zugfolge ausführen muß, um ein orientierungsfreies Teil zu orientieren.

Der Tetraederform kommt daher die Lösungsvariante aus [2] entgegen:

1. Drehe gesamtes MiniTet in eine bevorzugte Stellung
2. Bringe Puzzle in Tetraederform
3. Plaziere alle Spitzenteile
4. Orientiere diese
5. Plaziere die flachen Teile

Eine einfache Optimierung erhält man, wenn man aufeinanderfolgende Züge der gleichen Dimension zusammenfaßt oder ganz entfernt. Diese Situation kann beim Wechsel von einer Etappe zur nächsten auftreten.

MiniTet orientieren In diesem Schritt soll das MiniTet in eine für die weitere Lösung vorteilhafte Orientierung gebracht werden. Wir wollen das MiniTet so drehen, daß das Spitzenteil mit der Nummer 0 an Platz 0 mit der Orientierung 0 gesetzt wird. Bei allen weiteren Drehung soll dieses Teil nicht mehr bewegt werden. Das ist keine wirkliche Einschränkung, denn jede Drehung einer Scheibe, die den Platz 0 enthält, läßt sich durch eine Drehung der gegenüberliegenden Scheibe in entgegengesetzter Richtung ersetzen.

Der Algorithmus geht in drei Schritten vor:

1. Bringe Teil 0 an den Platz 0 oder einen gegenüberliegenden. Dazu wird einfach so lange um die z-Achse gedreht, bis dieser Fall eintritt.

2. Ist das Teil 0 jetzt am gegenüberliegenden Platz, wird es mit einer weiteren Vierteldrehung zum Platz 0 gebracht.

3. Die richtige Orientierung wird mit zwei weiteren Vierteldrehungen erreicht, wobei sich die Dimension der ersten Drehung frei wählen läßt. Nimmt man hier die der vorangehenden Drehung, läßt sich ein Zug einsparen. Somit läßt sich die gesuchte Orientierung für das MiniTet in höchstens drei Zügen herstellen.

Tetraederform herstellen Laut [2] reichen 3 Züge aus, um von jeder beliebigen Puzzlestellung die Tetraederform zu erreichen. Dieser Suchumfang könnte mit einer Tiefensuche problemlos bewältigt werden. Implementiert wurde aber ein Algorithmus, der gezielter sucht. Folgende Überlegungen sind dazu anzustellen:

Von welcher Form gelangt man mit nur einer Drehung zur Tetraederform, oder eine äquivalente Frage: welche Formen erreiche ich ausgehend von der Tetraederform? Man stellt fest, daß die Tetraederform erhalten bleibt, wenn man $180°$-Drehungen anwendet, und daß es nur eine weitere erreichbare Form (Schmetterlingsform, 2) gibt, die man mit $90°$-Drehungen herstellt. Wenn man jetzt eine Eigenschaft angeben kann, welche Tetraeder- und Schmetterlingsform eindeutig kennzeichnet, kann man die Tiefensuche auf zwei Züge beschränken.

Eine solche Eigenschaft lautet:

(A) In jeder Scheibe befinden sich genau zwei Spitzenteile.

Es ist klar, daß beide Scheiben einer Dimension genau dann jeweils zwei Spitzenteile besitzen, wenn das auf eine der beiden Scheiben zutrifft. Wenn sich in einer Scheibe zwei solche Teile befinden, können sie nur nebeneinander oder gegenüber liegend angeordnet sein. Davon ausgehend, überprüft man leicht, daß die genannte Eigenschaft auch hinreichend für Tetraeder- oder Schmetterlingsform ist.

Was zeichnet die Formen aus, die man von (2) aus erreicht? Von (2) ausgehend, ist die Drehebene die das Puzzle symmetrisch zerlegt, uninteressant, denn Drehungen in dieser Ebene führen nur zu (1) oder (2). Die anderen beiden Drehebenen unterscheiden sich in der Wirkung nicht, sie trennen das Puzzle in zwei Hälften, in denen jeweils zwei Eckenteile liegen, die zudem benachbart sind. Diese Eigenschaft bleibt beim Drehen erhalten.

(B) Puzzle läßt sich in zwei Scheiben zerlegen, in denen jeweils genau zwei Eckenteile auftreten und nebeneinanderliegen.

Man überzeugt sich leicht, daß (B) hinreichend für die Formen (2), (3), (4), (5) ist. Sollte weder (A) noch (B) erfüllt sein, liegt eine der Formen (6), (7) vor. Diese Formen überführt man in (3), (4) oder (5), in dem man eine Scheibe, die genau 1 oder 3 Spitzenteile enthält, um $90°$ dreht.

Algorithmus:

1. Test auf Form (1) → fertig

2. Test auf (A) → eine Vierteldrehung entlang der Symmetrieebene → (1) → fertig

3. Test auf (B) → drehe Ebene mit zwei benachbarten Spitzenteilen bis (2) hergestellt, weiter wie 2.

4. drehe Scheibe mit genau einem oder drei Spitzenteilen um $90°$, weiter wie 3.

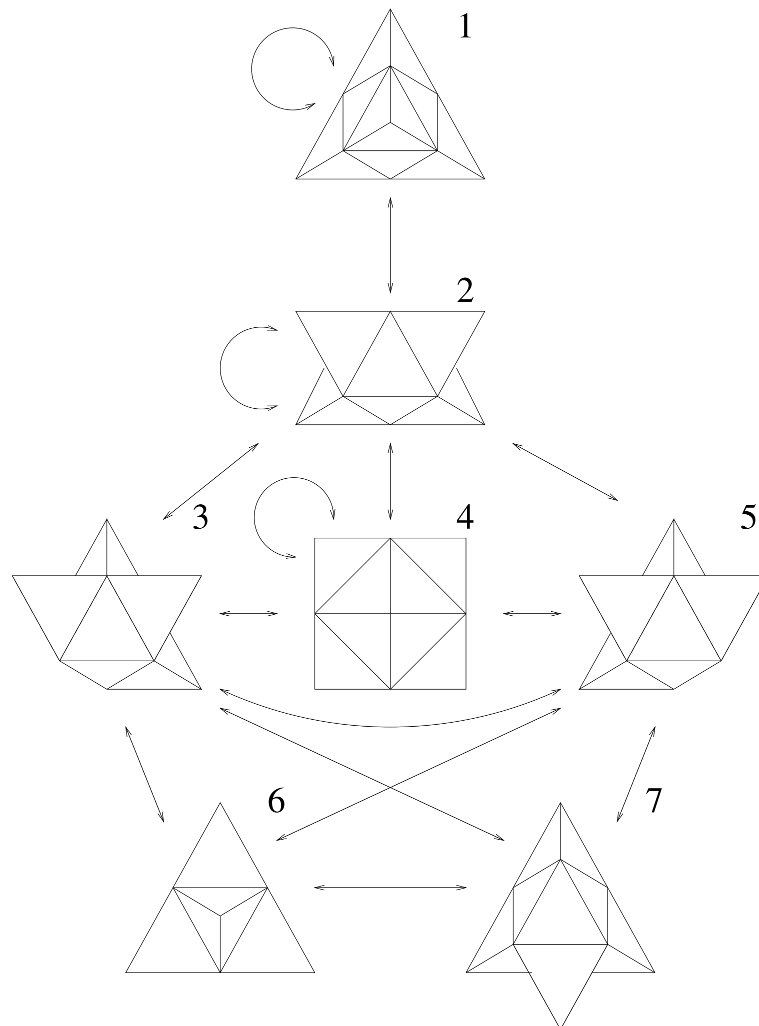


Abbildung 11: Mögliche Formen des Tetraeders und wie sie ineinander überführt werden können

Plazierung der Spitzenteile Da eines der Spitzenteile fixiert ist, können nur die anderen drei vertauscht sein (6 Permutationen). Wir betrachten nun einen weiteren Platz, an dem sich ein Spitzenteil befindet. Mit zwei Vergleichen stellen wir fest, ob sich dieses Teil am falschen Platz befindet, mit einer $180°$-Drehung ist dieses Prtoblem behoben. Dann können nur noch die verbleibenden zwei Spitzenteile vertauscht sein, was ebenfalls mit einer $180°$-Drehung gelöst wird.

Orientierung der Spitzenteile Alle zur Orientierung der Spitzenteile benötigten Zugfolgen aus [2] wurden im Programm fest gespeichert. Es sind nur fünf grundlegende Sequenzen, die sich nicht durch Spiegelung, Drehung oder Umkehrung ineinander überführen lassen. Allein mit Drehung und Umkehrung kann man aus dieser Menge alle Zugfolgen ableiten, die nötig sind, um mit einer Sequenz sämtliche Spitzenteilen zu orientieren. Das Rückwärts-Ausführen einer Folge bewirkt, daß alle positiven Orientierungsänderungen negativ werden und umgekehrt.

Zu jeder Zugfolge wird ein konstantes Feld angelegt, das für drei Ecken die Orientierungsänderungen einer Zugfolge enthält. In einem weiteren Feld wird die Zugfolge selbst durch Paare aus Drehdimension und Drehrichtung kodiert.

Das Programm probiert alle durch die Zugfolgen erreichbaren Orientierungsänderungen in jeder möglichen Drehung vorwärts und rückwärts durch. Sobald eine Orientierungsänderung alle Orientierungen aufhebt, wird die Zugfolge entsprechend abgewandelt angewandt.

Plazierung der flachen Teile Hierfür stehen vier Züge aus [2] zur Verfügung. Um jedoch für keinen Zug das Puzzle aus seiner Lage mit der fixierten Ecke bewegen zu müssen, werden darüberhinaus zwei daraus abgeleitete Züge fest im Programm abgelegt. Die Speicherung der Zugfolgen geschieht hier genau wie bei der Orientierung der Spitzenteile, etwas aufwendiger ist aber die Speicherung der Teilevertauschung. Da auch hier Züge rückwärts angewendet werden sollen, wäre es von Vorteil, wenn das Format für die Permutationen sowohl vorwärts wie auch rückwärts ausgelesen werden könnte.

Folgender Aufbau erfüllt diese Forderung: Die erste Zahl gibt den Platz eines Teiles an, die zweite Zahl gibt an, wohin das Teil vom ersten Platz verschoben werden soll. So wird mit den folgenden Zahlen fortgefahren. Taucht die Nummer des ersten Platzes des Zyklus nochmals auf, wird der Zyklus nach der damit verbundenen Verschiebeoperation beendet. Folgt in dem Feld eine weitere Zahl, leitet diese einen neuen Zyklus ein.

Da auch die Plazierung der flachen Teile mit einer einzigen Sequenz vollbracht werden soll, ist es wichtig, daß jeder Platz mindestens einmal in der Permutationsbeschreibung auftaucht, damit prüft das Programm automatisch ob dieser Platz das korrekte Teile enthält. Mögliche Alternative: Man sortiert die Zugfolgen so, daß die Zugfolgen, die viele Teile beeinflussen vor denen auf Anwendbarkeit überprüft werden, die wenig bewirken. Damit ist sichergestellt, daß das Programm Folgen vermeidet, die nur ein paar Teile richtig anordnen und andere übergehen.

Abstraktes Beispiel:

- Buchstabenfolge: abcdef
- Permutationsbeschreibung als Zahlenfeld: (2,3,4,2,1,6,1,5,5)
- Bedeutung: Vertausche Platz 2,3,4 zyklisch, Vertausche Platz 1 mit Platz 6, Teste Platz 5
- Ergebnis: fdbcea

3.1.5 Schnittstelle zur Grafik

Vom Programmteil, der die grafische Ein- und Ausgabe übernimmt, wird eine Beschreibung des Puzzles übergeben, die mit der internen Datenstruktur nicht ganz übereinstimmt. Da der Benutzer bei der Eingabe eine recht große Freiheit hat, ist es zum Beispiel möglich, daß man darin nicht existente Teile oder Teile mehrfach vorfindet.

Die Datenstruktur verwaltet die Teile in der gleichen Anordnung wie die Datenstruktur zum Lösen des Puzzles, der einzige Unterschied besteht in der Kodierung der Teile. Hier werden die Teile nicht

durch Nummer und Orientierung sondern durch ihre drei Farbblättchen kodiert. Die flachen Teile bekommen dreimal die gleiche Farbe. Die Routine zum Übertragen dieser Struktur in die Struktur für den Lösungsalgorithmus durchsucht für jedes Teil die Beschreibung aller Farbblättchen und versucht sie durch zyklisches Vertauschen in Übereinstimmung zu bringen. Die nötige Vertauschung gibt die Orientierung des Teils an. Ist diese Suche erfolglos, existiert das beschriebene Teil nicht. Im weiteren wird nur noch der Test durchgeführt, ob jedes Teil nur einmal verwendet wird (anhand eines Bitsets), womit sich der Test erübrigt, ob jedes Teil mindestens einmal eingesetzt wurde.

3.1.6 Automatische Tests

Die Algorithmen für das Herstellen der Tetraederform und das Plazieren der Spitzenteile wurden nur mit Hilfe von zufällig verdrehten MiniTets getestet. Für alle Zugfolgen der letzten beiden Lösungsetappen wurden Verdrehungen gefunden, die die gewünschten Zugfolgen erfordern. Damit wurde sowohl getestet, ob die richtige Zugfolge zur Anwendung ausgewählt wurde und ob die damit assoziierte Zugfolge richtig eingegeben war. In weiteren Tests wurde sichergestellt, daß Verdrehung und Umkehrung der gespeicherten Zugfolgen korrekt erfolgt. Zum Abschluß wurde der Lösungsalgorithmus einem ausführlichen Selbsttest unterzogen. Das Programm hat fortwährend mit zufälligen Zügen das Puzzle verdreht und danach mit dem Lösungsalgorithmus geordnet, und im Falle, daß das nicht gelingt, die Zugfolge ausgegeben, die zu der unlösbaren Stellung geführt hat und danach abgebrochen.

3.2 Rubiks Würfel

3.2.1 Theoretische Grundlagen

Zuerst soll untersucht werden, wie viele Möglichkeiten man beim Zusammenbau des Würfels hat. Die acht Ecken können beliebig permutiert und orientiert sein ($8! \cdot 3^8$), ebenso die Kanten ($12! \cdot 2^{12}$). Wir erhalten etwa $5.19 \cdot 10^{20}$ Zustände. Diese Zustände können nicht alle durch Drehen erreicht werden. Wir beobachten, daß es bei jeder Drehung folgende Invarianten gibt:

1. Die Summe der Orientierungen der Eckenwürfel bleibt gleich.
2. Die Summe der Orientierungen der Kantenwürfel bleibt gleich. Die genaue Festlegung der Orientierungen (siehe unten) ist dabei irrelevant.
3. Die Parität von Ecken- und Kantenvertauschungen zusammengenommen bleibt gleich. Erklärung: Führt man eine Drehung auf Vertauschung von Teilen zurück, dann ist die Gesamtzahl der Vertauschungen geradzahlig.

Bezeichnet man alle Zustände als äquivalent, die durch Drehungen ineinander überführbar sind, zerfällt die Menge aller Zustände in 12 gleich große Äquivalenzklassen (Universen, 3 mögliche Module der Eckenorientierungssummen $\times$ 2 mögliche Kantenorientierungssummenparitäten $\times$ 2 Vertauschungsparitäten). Die Mächtigkeit einer Äquivalenzklasse ist $5.19 \cdot 10^{20}/12 = 4.33 \cdot 10^{19}$. Man braucht mindestens 17 Außenscheibendrehungen um alle diese Stellungen von der Grundstellung aus zu erreichen ($18 \cdot 15^{15} = 7.88 \cdot 10^{18} < 4.33 \cdot 10^{19}$), maximal aber 52 mit dem Algorithmus von Thistlethwaite [3].

3.2.2 Testumgebung

Analog zur Lösung beim MiniTet wird zur Testausgabe und -eingabe eine Würfelbeschreibung verwendet, die die Seitenflächen widerspiegelt. Da diese alle möglichen Färbungen des Würfels (also auch unerlaubte) zuläßt und der grafischen Aufbereitung sehr nahe kommt, wird sie ebenfalls als Schnittstelle zum Grafikprogrammteil eingesetzt.

3.2.3 Datenstrukturen und Operationen

Die vom Lösungsalgorithmus verwendete Datenstruktur ist der vom MiniTet ganz ähnlich, nur das hier für jede der drei Teilesorten ein Feld mit Plazierungen und Orientierungen verwaltet wird. Für die Orientierungen der Ecken wird die gleiche Markierung verwendet wie beim MiniTet, die Mitten sind orientierungsfrei.

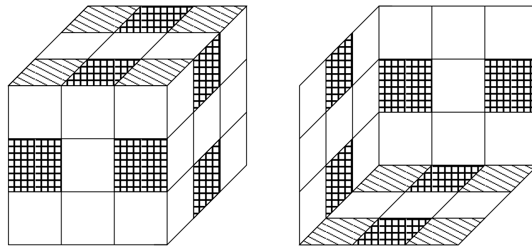


Abbildung 12: Codierung der Orientierung von Rubikwürfelteilen

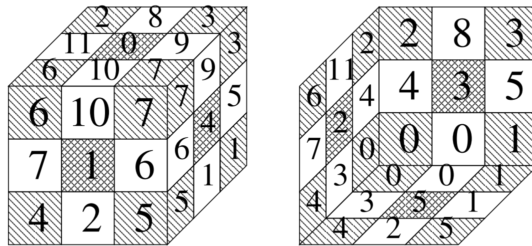


Abbildung 13: Nummerierung der Rubikwürfelteile

Bleibt noch zu klären, mit welchen Markierungen man die Orientierung der Kantenteile festmacht. Es hat sich als unpraktisch erwiesen, ebenfalls Grund- oder Deckfläche der unteren bzw. oberen Kantenteile zu markieren, denn diese Markierung läßt sich nicht auf die vier Kantenteile an der Seite fortsetzen. Stattdessen wurde eine Markierung gewählt, die gewährleistet, daß bei allen denkbaren Vierteldrehungen die Orientierungen sämtlicher beteiligter Kanten gekippt werden.

Im Gegensatz zum MiniTet ergibt sich die Schwierigkeit, daß sich die Kanten- und Mittenteile kaum logisch numerieren lassen, das heißt es ist schwer möglich, solche Fragen wie „Welche Teile gehören zu welcher Scheibe?“ allein mit einfachen Berechnungen zu beantworten. Es müssen daher Tabellen eingegeben werden, die verzeichnen, welche Permutationen durch eine Drehung bewirkt werden. Eine weitere Fehlerquelle, die durch Tests ausgeschaltet werden muß.

3.2.4 Lösungsalgorithmen

Wir hatten erkannt, daß anders als beim MiniTet nicht alle Permutationen aus der Grundstellung durch Drehungen erreichbar sind. Folglich muß vor dem Lösen sichergestellt werden, daß sich der angegebene Würfelzustand im gleichen Universum wie die Grundstellung befindet. Die Summe der Eckenorientierungen und die der Kantenorientierungen müssen 0 sein und die Inversionen von Ecken- und Kantenpermutation müssen gleiche Parität besitzen. Dieser Test wird jedem Lösungsversuch vorangestellt.

Vollständige Suche Die Anzahl der Stellungen des Rubik-Würfels übersteigt die der Stellungen des MiniTets um einige Größenordnungen. Eine Tiefen- oder Breitensuche zum Lösen des Puzzles ist daher völlig aussichtslos. Zu überlegen wäre, ob man durch die Wahl von geeigneten Kriterien Zwischenstationen definieren könnte, anhand derer man überprüfen könnte, ob die Suche noch in die richtige Richtung läuft. Ein solches Verfahren kann man zum Beispiel zum Ordnen der ersten Ebene einsetzen.

Intuitive Lösung Die Idee ist die gleiche wie beim MiniTet: Man zerlegt den gesamten Lösungsweg in mehrere Zwischenergebnisse und verwendet in jedem Schritt Zugfolgen, die die in vorigen Schritten erreichten Ergebnisse nicht wieder zerstören. Sowohl die Wahl der Etappen als auch die Komplexität der Zugfolgen, um eine Etappe zu bewältigen, läßt viele Möglichkeiten offen. Beim Thistlethwaite-Algorithmus werden die Zwischenstationen so gewählt, daß man in der ersten Etappe alle Züge verwenden darf, in der zweiten Etappe in einer bestimmten Dimension nur noch Doppeldrehungen, in der dritten Etappe bereits in

zwei Dimensionen nur noch Doppeldrehungen und in der letzten Etappe nur noch Doppeldrehungen, bei anderen Strategien werden zuerst alle Teile der Oberschicht, dann alle der Unterschicht und zum Schluß alle Teile der Mittelschicht geordnet, wieder andere Verfahren ordnen erst alle Ecken dann alle Kanten oder umgekehrt.

Hier wird das Verfahren aus [5] implementiert:

1. Drehe gesamten Würfel in eine bevorzugte Stellung
2. Plaziere Kanten der Oberschicht korrekt orientiert
3. Plaziere Ecken der Oberschicht korrekt orientiert
4. Plaziere Kanten der Mittelschicht korrekt orientiert
5. Plaziere Ecken der Unterschicht
6. Orientiere Ecken der Unterschicht
7. Plaziere Kanten der Unterschicht korrekt orientiert

Entweder man trifft für jede Phase eine kleinstmögliche Auswahl an Zugfolgen, die alle nötigen Operationen abdecken oder man benutzt Zugfolgen, die eine Phase im Ganzen lösen, in höchstens zwei Schritten oder auch mehr. Im ersteren Fall würden also nur einzelne Teile plaziert oder orientiert. Wir wissen aber, daß das nicht immer geht. Folglich kippt zum Beispiel eine Zugfolge, die eine Kante kippt, auch mindestens eine andere ungewollt. Ein Programm, das mit solchen Zugfolgen arbeiten kann, ist komplizierter, weil es manchmal die Nebenwirkungen von mehreren Zügen gegeneinander ausspielen muß. Die in [5] vorgestellten komplexen Züge haben diesen Mangel nicht, und führen gewöhnlich schneller zum Ziel, sind dafür wiederum schwerer zu merken, weil es deutlich mehr als unbedingt nötig sind.

Orientierung des Würfels Der Lösungsalgorithmus für den Würfel vereinfacht sich analog zu dem für das MiniTet, wenn man eine bestimmte Orientierung des gesamten Würfels voraussetzen kann. Da die Mittelteile des Würfels durch ein Gestell fest miteinander verbunden sind, bietet sich die Fixierung der Mittelteile gegenüber der Fixierung eines Eckenteiles an.

So funktioniert der Algorithmus:

1. Es wird überprüft, ob am Würfel alle Mittelteile, die sich gegenüberliegen müssen, das tatsächlich tun.
2. Das Mittelteil 0 wird aufgesucht und mit einer Drehung des ganzen Würfels an seinen Platz gebracht.
3. Der Würfel wird so lange um die senkrechte Achse gedreht (die Achse senkrecht durch das Mittelteil 0) bis Teil 1 an der richtigen Stelle ist. Diese Schleife terminiert, da dieses Teil tatsächlich eingesetzt wurde, sonst wäre ein anderes Mittelteil doppelt verwendet worden und das wäre in einem früheren Test aufgefallen und zum anderen muß sich wegen 1. dieses Teil außerdem auf dem Äquator befinden.
4. Test, ob Position 2 das Teil 2 enthält. Dieser Test stellt sicher, daß die Mittelteile 0, 1, 2 in der richtigen Position zueinander stehen. Wegen 1. sind dann alle Teile korrekt plaziert.

Plazierung der Kanten in der Oberschicht Die Anzahl der Konstellationen bei diesem Problem ist recht groß und die Längen der Zugfolgen gering. Es läßt sich nachprüfen, daß stets vier Züge genügen, um eine weitere Kante in der oberen Ebene korrekt orientiert einzusetzen. Es wäre somit unklug, eine vollständige Fallunterscheidung zu führen und womöglich auf besonders elegante Zugfolgen zu verzichten, die man mit der vollständigen Suche im allgemeinen findet.

Die ganze Etappe läßt sich mit einer Tiefensuche zwar nicht lösen, aber einzelne Schritte schon. Im schlechtesten Fall soll der Suchalgorithmus jedes Kantenteil einzeln einordnen, wir wollen die Möglichkeit

aber nicht ausschließen, bessere Zugfolgen zu finden. Dazu werden alle Zugfolgen bis zu einer vorgegebenen Länge untersucht und es wird die Sequenz angewendet, die die beste Bewertung erhält. Bei gleichen Bewertungen hat natürlich die kürzere Lösung Vorrang.

Wir führen ein Bewertungssystem ein, das dem Suchalgorithmus mitteilt, ob bereits eine Verbesserung erreicht wurde und wie gut diese gegenüber anderen ist. Gezählt wird die Anzahl der korrekt einsortierten Kantenteile. War vor der Suche zum Beispiel eine Kante korrekt einsortiert, stellen zwei korrekte Kanten eine Verbesserung dar, aber eine Zugfolge, die sogar drei korrekte Kanten einordnet, ist natürlich zu bevorzugen.

Die so geführte Bewertung unterscheidet nicht, welche korrekt einsortierten Kanten hinzugekommen sind. Es wäre durchaus denkbar, daß eine korrekte Kante wieder entfernt wird, um dafür die drei anderen Kanten einzusortieren.

Wenn wir die Suchtiefe mit vier Zügen ansetzen, ist eine Lösung garantiert und der Algorithmus führt mit Sicherheit zum Ziel. Längere Zugfolgen lassen sich derzeit nicht in akzeptabler Geschwindigkeit finden.

Plazierung der Ecken in der Oberschicht Die Technik ist genau die gleiche, nur das Punktesystem weicht ab. Das Problem beim Einsortieren der Ecken ist, daß schon die Zugfolgen zum Einsortieren einzelner Teile bis zu sechs Züge beanspruchen - zu lang für eine Tiefensuche. Problemfälle sind Ecken, die mit der Deckseite nach unten in der Unterschicht liegen und Ecken die falsch in der Oberschicht angeordnet sind. Es reicht daher nicht, Teile zu bewerten die endgültig eingeordnet sind, es müssen auch die beachtet werden, die in wenigen Zügen einsortiert werden können. Andererseits soll es sich für den Suchalgorithmus nicht lohnen, schon korrekt einsortierte Ecken wieder aus der Oberschicht zu nehmen, um sie in leicht plazierbare zu verwandeln.

In der Praxis hat sich folgendes Punktesystem bewährt:

- 3 Punkte für jede korrekt plazierte Ecke
- 1 Punkt für jede Ecke in der Unterschicht, deren Oberseite nicht nach unten zeigt

Folgerung: Nur wenn durch das Entfernen einer schon korrekt plazierten Ecke mindestens drei leicht plazierbare Ecken entstehen, kann dieser Tausch akzeptiert werden. Die Bewertung arbeitet recht vorausschauend, denn von mehreren leicht plazierbaren Ecken wird immer die einsortiert, die beim Plazieren keine andere Ecke in eine ungünstige Position bringt.

Auch hier kann man überprüfen, daß eine Bewertungssteigerung innerhalb von vier Zügen immer möglich ist.

Lösung der restlichen Etappen Im Gegensatz zur MiniTet-Lösung wird nicht für jede Etappe eine eigene Routine verwendet. Stattdessen werden die Zugfolgen aller Etappen in einer sortierten Liste gespeichert. Die Sortierkriterien in absteigender Priorität:

1. Frühe Etappe
2. große Auswirkung (viele bewegte Teile)
3. geringe Zugzahl

Das erste Kriterium stellt sicher, daß jede Etappe abgeschlossen wird, bevor zur nächsten übergangen wird. Das zweite Kriterium garantiert, daß Züge mit komplexer Wirkung nicht durch einfachere ersetzt werden (was zu einer höheren Zugzahl führen würde). Dem dritte Kriterium liegt die Idee der greedy-Algorithmen zugrunde, nur daß hierbei keineswegs sicher ist, daß diese Wahl zum ingesamt kürzeren Ergebnis führt, weil ein einfacher Zug zur falschen Zeit die nachfolgenden Schritte deutlich verkomplizieren kann.

Wie beim MiniTet auch besteht die Beschreibung jeder Zugfolge aus ihrer Wirkung und der Zugfolge selbst. Die Wirkung besteht hierbei aus der Kanten- und Eckenpermutation, in der Regel ist nur eine von beiden besetzt. Die Permutationsbeschreibung enthält ebenfalls die Orientierungsänderung jedes bewegten Teiles und es ist möglich anzuzeigen, daß die korrekte Orientierung oder Plazierung durch diese Sequenz noch nicht erreicht werden muß.

Die Lösungsroutine geht nun die Datenbank durch, überprüft ob die jeweiligen Wirkungen alle vorgesehenen Teile richtig plazieren und orientieren, oder ob man dies durch Spiegeln oder Drehen um die senkrechte Achse oder durch Voranstellen eines Zubringerzuges in der unteren Ebene erreicht. Wird eine solche Sequenz gefunden, wird sie entsprechend modifiziert und angewendet. Ist keine Zugfolge mehr anwendbar, geht der Algorithmus davon aus, daß das Puzzle gelöst wurde. Die Datenbank muß für jede Zugfolge von vorne (oder zumindest vom Anfang der Etappe) durchsucht werden, da die Anwendung einer Zugfolge womöglich die Vorlage für eine andere schafft, die in der Sequenz-Liste vor der aktuellen liegt.

3.2.5 Überführung unterschiedlicher Stellungen

Da es beim Rubik-Würfel keine Veränderungen der Puzzle-Form gibt, gestaltet es sich einfacher, den intuitiven Lösungsalgorithmus zum Überführen zweier von der Grundstellung verschiedenen Stellungen einzusetzen.

Auf der Hand liegt die doppelte Anwendung der Grundlösung: Man löst von der Ausgangsstellung in Richtung Grundstellung, dann von der Zielstellung in Richtung Grundstellung und setzt den zweiten Lösungsweg invertiert an den ersten. Ergebnis ist ein durchschnittlich doppelt so langer Lösungsweg, der darüberhinaus noch einen Nachteil hat: Man kann zwei Stellungen nicht ineinander überführen, wenn sie zwar dem gleichen Universum, nicht aber dem Universum der Grundstellung angehören. Es wäre ja denkbar, daß jemand mit einem falsch zusammengebauten Würfel arbeitet und zwei darauf realisierbare Stellungen ineinander überführen möchte.

Wir wollen versuchen, das erweiterte Problem direkt auf den Grundalgorithmus zurückzuführen. Dazu betrachten wir jede Stellung des Würfels als Permutation seiner Teile, ebenso betrachten wir jeden Zug und jede Zugfolge als Teilepermutation.

- O sei die Permutation des Grundzustandes
- A sei die Ausgangsstellung
- B sei die Zielstellung
- p sei die Permutation die A in B überführt, also $A \cdot p = B$

Gesucht ist eine Faktorisierung von p in Einzelzüge. Der ursprüngliche Algorithmus faktorisiert p, wenn $X \cdot p = O$ gilt und man dem Algorithmus X als Ausgangsstellung übergibt. Also formen wir um:
$A \cdot p = B$
$B^{-1} \cdot A \cdot p = O$
Wir müssen dem Lösungsalgorithmus $B^{-1} \cdot A$ übergeben und erhalten direkt die Lösung für das abgewandelte Problem.

Wie berechnet man $B^{-1} \cdot A$?
Man wendet die Permutation A auf B^{-1} an. B^{-1} bestimmt man durch Auflösen von $Y \cdot B = O$ nach Y.
Eine Permutation sei durch eine Tabelle gegeben, welche von den Originalplätzen auf die Zielplätze abbildet.
Invertierung von B ($Z = B^{-1}$) bedeutet: Für jedes n nimm n-te Zahl aus B, beschreibe den dadurch adressierten Eintrag in Z mit n.
Anwendung von A auf X ($Z = X \cdot A$) bedeutet: Für jedes n nimm n-te Zahl aus X, schreibe sie in den Eintrag von Z, der durch die n-te Zahl in A adressiert wird.

3.2.6 Automatische Tests

Um Tippfehler beim Erstellen der Zugfolgen-Datenbank aufzuspüren, wurde die sortierte Liste bei den Zugfolgen der letzten Etappe beginnend abgearbeitet und jede Zugfolge wurde auf den anfangs sortierten Würfel angewendet. Danach wurde verglichen, ob die Zugfolgen einer Etappe tatsächlich alle Teile, die sie nicht verändern dürfen, unberührt ließen.

Der zweite Test sollte überprüfen, ob die Permutationen korrekt eingegeben sind, in dem die zugehörige Zugfolge rückwärts ausgeführt wurde und dann der Lösungsalgorithmus auf den erhaltenen Zustand angewendet wurde. Dieser Test läuft leider nicht selbständig, denn manche Zugfolgen erzeugen beim Rückwärtsausführen eine Situation, die sich auch anders lösen läßt.

Zum Schluß erfolgte der Routine-Test, der den Würfel zufällig verdreht und anschließend selbständig löst. Versagt der Lösungsalgorithmus bricht der Test ab.

4 Literatur

[1] Rainer Barth, Ekkehard Beier, Bettina Pahnke: Grafikprogrammierung mit OpenGL, Addison-Wesley, 1996

[2] Hans-Dietrich Gronau: Kombinatorische Spiele im Raum - Das magische Tetraeder, in: alpha August 1997, Velten 1997

[3] Wolfgang Hintze: Der ungarische Zauberwürfel, Berlin 1986

[4] Wolfgang Hintze: Die Verwandten des Zauberwürfels, Berlin 1985

[5] Wolfgang Schifferdecker: Der Dreh mit dem Würfel, Leipzig 1982