QCrystalTile

Contents

Ermittlung und Speicherung der Space Group Informationen

Hilfsmittel

JSON(JavaScript Object Notation):

JSON ist ein Datenformat, in dem Informationen wie Arrays, Objekte etc. in lesbarer Form gespeichert werden und bei bedarf wieder erzeugt werden können. Die Daten werden dabei sprachunabhängig gespeichert und können somit auch sprachübergreifend genutzt werden. Parser existieren in fast allen verfügbaren Programmiersprachen.

Ermittlung & Speicherung:

Aus Effizienzgründen wurden die Informationen der einzelnen Raumgruppen in einem JSON-Array lokal gespeichert. Die Informationen sowie die Transformationen der Raumgruppen haben wir von folgender Seite extrahiert:

http://homepage.univie.ac.at/nikos.pinotsis/spacegroup.html

Für eine übersichtlichere Nutzung wurde die html-Datei geparst und überflüssige Formatierungen entfernt. Die Daten entsprachen nach dem Bearbeiten der einheitlichen Form:

- Space Group Name = P1
- Crystal System = TRICLINIC
- Laue Class = -1
- Point Group = 1
- Patterson Space Group # = 2
- Lattice Type = P
- symmetry= X,Y,Z

wodurch die Daten leicht getrennt und nach unseren Bedürfnissen als JSON-Array gespeichert werden konnten.

SpaceGroupFactory

Die SpaceGroupFactory wird benutzt um die benötigte Raumgruppe aus der lokal gespeicherten JSON-Datei zu erzeugen. Dazu wird die Raumgruppe über ihre ID identifiziert und anhand ihrer Daten wird ein SpaceGroup-Object erstellt. Schwierigkeiten hierbei gab es bei den Transformationen, da diese als String in Koordinatentransformationen gespeichert sind:

Bsp: X, 1/2 +Y, Z

wir aber mit 4x4 Matrizen arbeiten.

```
X   0   0   0
0   Y   0   1/2
0   0   Z   0
0   0   0   1
```

Um die Koordinatentransformationen in die 4*4 Matrizen umzuwandeln wurden die Transformationen jeweils geparst und dann in Zeilenvektoren umgewandelt.

Dabei war die einheitliche Form der Transformationen ein großer Vorteil, da Konstante, Operator und Variable immer in einer festen Reihenfolge angeordnet waren und somit leichter getrennt werden konnten. Aus den Zeilenvektoren konnten nun die Transformationen in 4*4 Matrizen dargestellt werden.

Gruppierung der Raumgruppen

Die SpaceGroupFactory bietet neben dem Erstellen der Raumgruppen noch die Möglichkeit die Raumgruppen nach bestimmten Kriterien auszuwählen (Kristallsystem, Zentrierung). Dabei wird wieder anhand der gesuchten Kriterien durch das JSON-Array iteriert und ein Set der angeforderten Transformationen zurückgegeben.

Mathematische Vorraussetzungen und Konkretisierung des zu visualisierenden Sachverhalts

Vorbemerkung: mit welchen Polyedern kann man den Raum pflastern?

Die von uns entwickelte Software dient der Visualisierung eines Problems der Geometrie. Dabei geht es um die Frage: Mit welchen Körpern kann man lückenlos den Raum füllen? Je nachdem kann man dieses Problem auch mit dem einen oder anderen Schwerpunkt untersuchen, z.B. wird auf "mathoverflow"[1] folgendes gefragt:

- Welches ist der komplexeste Körper, mit dem man den Raum füllen kann?
- Im Falle, dass dieser Körper eine Polyeder ist: Wie viele Ecken/Kanten/Flächen kann dieses Polyeder maximal haben?

Es gibt eine Vielzahl von Arbeiten über Probleme dieser Art. Im allgemeinen sind derartige Fragestellungen beliebig komplex. Für eine bestimmte Klasse solchen Raumfüllungen, oder Parkettierungen sind allerdings Lösungen bekannt.

Um welche Art von Parkettierungen geht es?

Unsere Software dient der Visualisierung einer bestimmten Klasse von Parkettierungen. Zunächst soll also skizziert werden, um welche Art von Parkettierungen es hier geht und wie diese konstruiert werden können. (Die folgenden Ausführungen stützen sich auf den Artikel von Engel[2])

Die Elemente (im folgenden “Kacheln”) dieser Parkettierung sollen folgende Eigenschaften haben:

- es soll sich um *konvexe* Polyeder handeln
- Alle Kacheln sollen kongruent oder spiegelbildlich kongruent sein
- die Kacheln sollen nur mit ganzen Flächen aneinanderstoßen
- die entstehende Raumteilung soll *“homogen”* sein. (das heißt: es gibt eine Symmetriegruppe, unter der alle Kacheln kongruent sind)

[1] “Foreneintrag zur diskussion mit dem thema: how many vertices/edges/faces at most for a convex polyhedron that tiles space?”, http://mathoverflow.net/questions/86042/how-many-vertices-edges-faces-at-most-for-a-convex-polyhedron-that-tiles-space/86056#86056.

[2] “Über wirkungsbereichsteilungen von kubischer symmetrie von peter engel, zeitschrift für kristallographie, volume 154 (1981), number 3–4, 199–215”, http://www.degruyter.com/view/j/zkri.

Zusätzlich wird gefordert, dass die Raumteilung *"homogen"* (auch "regulär" oder "isohedral") sei.

Der letzte Punkt fordert also eine Symmetriegruppe. Für diese gilt:

- da alle Kacheln unter ihr kongruent sind, können aus einer einzigen Kachel durch Anwendung dieser Gruppe alle Kacheln der Parkettierung erzeugt werden
- Da die Kacheln den Raum vollständig bedecken sollen, muss diese jene Symmetriegruppe Translationen beinhalten. Das heißt es handelt sich bei dieser Symmetriegruppe um eine sogenannte *Raumgruppe*

In dem Artikel wird gezeigt, dass eine solche Parkettierung als "Wirkungsbereichsteilung" ("Voronoi-Diagramm") einer homogenen Punktmenge entsteht.

Dabei ist die Symmetriegruppe der Punktmenge die gleiche wie die der "Wirkungsbereichsteilung" der Punktmenge.

"homogenes Punktsystem" meint dabei eine Punktmenge, die unter einer Symmetriegruppe isomorph ist.

Insgesamt also: Es gibt eine Symmetriegruppe, genauer, eine Raumgruppe $\boldsymbol{G}$, und eine Punktmenge $\boldsymbol{M}$, so dass sowohl $\boldsymbol{M}$ als auch das Voronoi-Diagramm unter $\boldsymbol{G}$ isomorph sind.

Konstruktion einer Parkettierung

Eine Parkettierung mit den oben geforderten Eigenschaften lässt sich also folgendermaßen konstruieren:

1. konstruiere die homogene Punktmenge $\boldsymbol{M}$:
    1. wähle einen Punkt $\boldsymbol{P}$
    2. wähle eine Raumgruppe $\boldsymbol{G}$
    3. Sei $\boldsymbol{M}$ die Menge aller Bilder von $\boldsymbol{P}$ unter $\boldsymbol{G}$
2. Sei $\boldsymbol{V}$ die Voronoi-Zerlegung der Punktmenge $\boldsymbol{M}$

wegen den obigen Ausführungen gilt jetzt: $\boldsymbol{V}$ ist jetzt eine Parkettierung mit den geforderten Eigenschaften

Meta-Design

Das Programm soll Parkettierungen des anfangs definierten Typs anschaulich darstellen können. Dazu muss es auch in der Lage sein, Ausschnitte solcher Parkettierungen zu berechnen. Zuletzt muss die Berechnete Parkettierung (= Voronoi-Zerlegung) visualisiert werden. Anschaulich gesprochen muss dazu folgende Kette von Verarbeitungs-Schritten realisiert werden:

```
Punkt P, Raumgruppe G -> Punktmenge M -> Voronoi-Zerlegung V -> Visualisierung
```

Dies legt eine Filter-Architektur nahe, jeder Berechnungs-Schritt (im Diagramm durch einen Pfeil dargestellt) dieser Kette entspricht einem Filter.

Dieser Ansatz ignoriert allerdings Aspekte, die jenseits dieses Datenflusses liegen:

- Wann wird diese Verarbeitungs-Kette angestoßen?
- Die Visualisierung wird ein GUI benutzen. Wie ist dieses in der Lage, die Verarbeitungs-Kette mit vom User geänderten Parametern neu anzustoßen?

Das Model-View-Controller-Pattern (im folgenden "MVC-Pattern") ist eine bewährte Strategie diesen Fragen zu begegnen. Unser Design besteht in einer Verschmelzung einer Filter-Architektur mit dem MVC-Pattern. Dazu werden die einzelnen Berechnungs-Schritte der obigen Verarbeitungs-Kette wie folgt auf die drei Komponenten des MVC-Patterns verteilt:

```
Punkt P, Raumgruppe G ->(1) Punktmenge M ->(2) Voronoi-Zerlegung V
 ->(3) Visualisierung

Model: (1):
    Punkt P, Raumgruppe G -> Punktmenge M
View: (2, 3)
    Punktmenge M -> Voronoi-Zerlegung V -> Visualisierung
Controller:
    - erzeugt, und verknüpft Model und View
```

Kommentar: Das Datenmodell besteht in den zu visualisierenden Daten. Diese bestehen in diesem Fall eigentlich aus einer bestimmten Parkettierung V. Da die Parkettierung allerdings durch die Wahl der Raumgruppe G, sowie eines einzigen Punktes P eindeutig festgelegt ist, genügt es, diese 2 Parameter als Datenmodell zu wählen. Die Voronoi-Zerlegung wird in die Darstellung des M-V-C-Patterns verschoben. Dadurch ist das Datenmodell für das Ergebnis des Algorithmus (nämlich die Parkettierung V) eine Frage der "View".

Durch diese Überlegungen ist eine Art Meta-Design festgelegt, das 3 Module, nämlich Model, View und Controller Vorschlägt, deren innere Struktur jedoch

offen lässt. Selbst die genauen Schnittstellen zwischen diesen Modulen, können zunächst vage bleiben, und ergeben sich automatisch während des Entwicklungsprozesses.

Die folgenden Kapitel geben einen Einblick sowohl in das Design, als auch die Implementierung der Innenstruktur dieser 3 Komponenten.

Design und Implementierung des Models

Im folgenden werden die wichtigsten Entscheidungen bei Design und Implementierung des Models dargelegt. Dabei werden teilweise Vereinfachungen vorgenommen und unwichtige Aspekte zu Gunsten der Übersichtlichkeit übergangen.

Design des Models

Ausgehend von der Spezifikation müssen die Parameter des Parkettierungs-Algorithmus in Klassen umgesetzt werden. Das sind:

- gewählter Punkt $\boldsymbol{P}$
- Punktmenge $\boldsymbol{M}$
- Raumgruppe $\boldsymbol{G}$

Repräsentation von Vektoren und Matrizen: la4j

Die Verwendung einer Software-Library für Lineare Algebra, nämlich "la4j"[3] legt die folgende Umsetzung nahe:

- Punkt $\boldsymbol{P}$ -> Vector3D
- Punktmenge $\boldsymbol{M}$ -> Set

Vector3D ist dabei eine Klasse, die die Funktionalität der Klasse `Vector` von la4j *** kapselt. Die Benutzung eines solchen Proxies erlaubt es, von der verwendeten Software-Bibliothek zu abstrahieren (Proxy pattern[4]). Analog werden auch Proxies `Matrix3D`, `Matrix4D` definiert.

UML-Diagram der wichtigen Klassen des Models

Das folgende UML-Diagram zeigt die Innenarchitektur des Models. Die folgenden Kapitel kommentieren die wichtigsten Überlegungen, die zu diesem Design geführt haben.

[3] "Linear algebra for java", http://la4j.org/.
[4] "Wikipedia", http://en.wikipedia.org/wiki/Proxy_pattern.

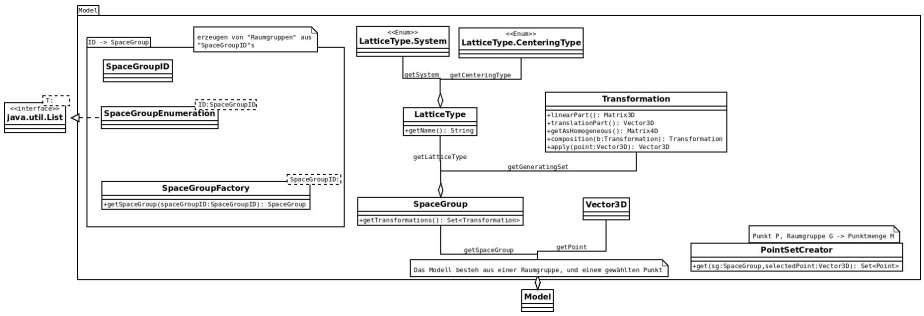


Figure 1: “Innenarchitektur des Model”

Repräsentation einer Raumgruppe

Weniger naheliegend ist die Frage, wie eine Raumgruppe in Java modelliert werden kann. Gesucht ist dabei eine Darstellung, die eine effiziente Berechnung der homogenen Punktmenge aus dem Punkt $\boldsymbol{P}$ ermöglicht. Hierfür sollte es möglich sein, die Transformationen der Gruppe aufzulisten. Dies legt nahe, eine Raumgruppe als *Menge von Transformationen* zu modellieren.

Für die konkrete Implementierung ist es sinnvoll, die entsprechenden Klasse um weitere Informationen anzureichern:

- ‘`SpaceGroup.getLatticeType`‘: dieses Feld steht für den Gittertyp der Raumgruppe
- ‘`SpaceGroup.getGeneratingSet`‘: eine Menge von Transformationen, die die Raumgruppe erzeuge

`SpaceGroup.getTransformations` gibt die Menge der Transformationen in der Raumgruppe zurück. Dies sind theoretisch unendlich viele. Da die Visualisierung nur einen endlichen Ausschnitt der Raumteilung darstellen kann, lässt sich diese Menge allerdings einschränken, indem alle Transformationen weg gelassen werden, die den Punkt P aus dem zu visualisierenden Bereich transformieren würden. In der Tat genügt es, alle Transformationen zu erzeugen, die den Punkt nicht aus der Einheitszelle heraustransformieren (siehe Kapitel: Implementierung der Klasse `SpaceGroup`)

Die Erzeugung einer Implementierung der Klasse `SpaceGroup` wurde gemäß dem Factory-Pattern in die Klasse `SpaceGroupFactory` ausgelagert. Die zugehörigen Klassen sind im Diagramm als eigenes Paket dargestell (`ID -> SpaceGroup`).

Repräsentation der Transformationen Oben haben wir eine Raumgruppe als Menge von Transformationen modelliert. Nun soll es um die Frage gehen, wie Transformationen im Programm repräsentiert werden.

Welche Art von Transformationen? Bei den Transformationen handelt es sich um Verschiebungen (d.h.) die als Komposition von Transformationen folgenden Typs entstehen:

- Translationen
- Rotationen
- Roto-Inversionen

naheliegende Möglichkeiten der Darstellung von Transformationen Für die Darstellung der Transformationen gibt es folgende naheliegende Möglichkeiten:

- *als Matrizen*: Rotationen, sowie Roto-Inversionen können durch 3x3-Matrizen kodiert werden. Translationen, sowie Mischformen von Translationen und Rotationen, lassen sich zwar auf diese weise nicht direkt darstellen, wohl aber mittels sogenannter *homogener Koordinaten*. Statt 3x3- werden dabei 4x4-Matrizen verwendet.

  Sei rMatr eine 3x3-Rotations-Matrix, (sx, sy, sz) ein TranslationsVektor. Dann sind die zugehörigen homogenen 4x4-Matrizen wie folgt definiert:

```
homRot   =  ( rMatr11 rMatr12 rMatr13 0 )
            ( rMatr21 rMatr22 rMatr23 0 )
            ( rMatr31 rMatr32 rMatr33 0 )
            ( 0       0       0       1 )

homTrans =  ( 1 0 0 sx )
            ( 0 1 0 sy )
            ( 0 0 1 sz )
            ( 0 0 0 1  )
```

- *als Quaternionen*: Tatsächlich lassen sich Rotationen auch als Quaternionen darstellen. Dabei handelt es sich um eine Verallgemeinerung des Konzeptes der Komplexen Zahlen. Der Translations-Anteil einer Transformation müsste separat kodiert werden, z.B. als 3-dimensionaler Vektor.

Für die Berechnung der Punktmenge M sind Matrizen aus folgenden Gründen optimal:

- Verkettung von Transformationen, sowie Anwenden einer Transformation auf den Punkt entspricht einer Matrixmultiplikation. In der Software-Bibliothek la4j ist diese Bereits implementiert.
- Matrizenrechnung ist ein gängiges mathematisches Konzept, was der Wartbarkeit der Software zu Gute kommt und sowohl für beim Implementieren als auch beim Debugging von Vorteil ist.

Abstraktion der internen Darstellung über die Klasse `Transformation`
Naheliegend wäre es also, Transformationen als 4x4-Matrizen zu kodieren. Stattdessen haben wir uns entschieden, eine eigene Klasse für Transformationen bereit zu stellen. Die eigentliche interne Darstellung ist somit geheim, und kann beliebig ausgetauscht werden. Die Möglichkeit, Verkettungen von Transformationen in einer Methode zu kapseln hat entscheidende Vorteile, wie sich herausgestellt hat.

Es ist an dieser Stelle wichtig, dass für die Komposition von Transformationen immer die Methode `Transformation.compose` benutzt wird, und nicht die Matrixmultiplikation! Ansonsten ist das Verhalten nicht spezifiziert.

An die Implementierung der Transformationen stellen wir 2 Forderungen:

1. Sei `tn` eine Transformation, die der n-fachen Rotation um den Ursprung entspricht. Dann

   tn * tn * ... * tn = id

2. Sei `s` eine Transformation, die einer Translation entspricht. Dann

   s - s = id

(beide Punkte werden im folgenden unter dem Begriff **Kompositionseindeutigkeit** zusammengefasst).

Interface nach außen

Nachdem nun die wichtigen Datentypen des Models spezifiziert sind, soll hier skizziert werden, wie das Model von anderen Modulen benutzt werden kann, um eine homogene Punktmenge M zu berechnen:

```
// erzeugen einer Raumgruppe:
SpaceGroupFactory factory = new SpaceGroupFactoryImpl();
SpaceGroup spaceGroup = factory.getSpaceGroup( new IDImpl("P1"));

// wählen eines Punktes:
Vector3D point = new Vector3D( new double[] { 0.1, 0.1, 0.1 } );

// erzeugen der homogenen Punktmenge:
PointSetCreator pointSetCreator = new PointSetCreatorImpl();
Set<Vector3D> pointSet = pointSetCreator.get( spaceGroup, point );
```

Implementierung des Datenmodells

Implementierung der Klasse `SpaceGroup`

Für die Funktionalität der Implementierung von `SpaceGroup` ist hauptsächlich die Implementierung des Interfaces `Transformation` entscheidend, und die Implementierung eine Herausforderung (siehe unten). Die einzig interessante Aufgabe ist das Implementieren der Methode `SpaceGroup.getTransformations`. Diese Methode soll aus einer kleinen Menge von Transformationen weitere Transformationen berechnen.

sei E die Menge der Erzeuger-Transformationen.

```
H = E

{
    H' = H
    für alle e in E:
    für alle h in H:
    {
        neu = e * h
        if inEinheitsZelle( neu )
        {
            füge neu zu H' hinzu
        }
    }
    H = H'
} wiederhole, falls |H| größer geworden ist
```

Dieser Algorithmus berechnet alle Transformationen, die als Komposition der Erzeuger-Transformationen vorkommen, und nicht aus der Einheitszelle herausführen. Die Bedingung `inEinheitsZelle` zu implementieren, ist tatsächlich nicht ganz einfach, soll zu Gunsten der Übersichtlichkeit hier nicht erklärt werden.

Implementierung der Klasse `PointSetCreator`

Pseudocode:

```
sei ‘point‘ der gewählte Punkt
sei ‘spaceGroup‘ die gewählte Raumgruppe

für alle transformation in spaceGroup.getTransformations:
{
    füge transformation.apply( point ) zu M hinzu
}
```

Dieser Algorithmus erzeugt den Ausschnitt aus der homogenen Punktmenge M, so dass alle Punkte in der Einheitszelle liegen. Soll ein größerer Ausschnitt berechnet werden, so müssen zu dem Resultat auch alle Translationen der Punktmenge hinzu, die in dem gewünschten Ausschnitt liegen.

Implementierung der Klasse `Transformation`

Es wäre naheliegend, auch für die interne Darstellung der Transformationen 4x4-Matrizen zu verwenden.

Dies hat allerdings einen fatalen Nachteil:

Die Komposition dieser Implementierung erfüllt nicht die Forderung nach **Kompositionseindeutigkeit** (siehe oben).

Das Problem ist, dass in dieser naiven Implementierung die Komposition von Transformationen über eine Matrix-Multiplikation bewerkstelligt wird. Die Rundungsfehler der Fließkommazahlenrechnung machen diesen Algorithmus zu instabil um **Kompositionseindeutigkeit** garantieren zu können.

dieses Problem zu umgehen kann man sich zu Nutze machen, dass in Raumgruppen auf Grund der ???kristallographischen Restriktion??? sowohl Rotationen als auch Translationen nicht in beliebig kleinen Schritten vorkommen. Durch Runden des Ergebnisses auf den ggT dieser aller Rotationen bzw. Translationen kann dieses Problem umgangen werden. Dieses Prinzip des Rundens ist allerdings in der Matrix-Darstellung schwer zu bewerkstelligen. Für diesen Zweck wird intern eine andere Darstellung für Transformationen verwendet:

interne Darstellung von Transformationen Vereinfacht sieht die interne Darstellung einer Transformation $\boldsymbol{t}$ so aus:

```
t:
    Euler-Winkel r = (rx, ry, rz)
    Translation s = (sx, sy, sz)
    Spiegelungs-Matrix oder EinheitsMatrix m
```

Das heißt die Transformation wird in einen linearen Teil, nämlich eine Rotation und eine Translation aufgeteilt. Die Matrix $\boldsymbol{m}$ wird benötigt, um Roto-Inversionen darstellen zu können. Handelt es sich um eine pure Rotation, so sei m die Einheitsmatrix, falls nicht die Matrix der Punktspiegelung am Ursprung. Die Rotation wird aber *nicht* als eine Rotationsmatrix dargestellt, sondern in eine Folge von 3 Rotationen um die x-, y-, und z-Achse (die 3 Komponenten des Vektors `rotVec`) aufgeteilt. Diese 3 Komponenten werden auch *Euler-Winkel* genannt.

Gemeint sind 3 Winkel, so dass für die 3x3-Matrix $\boldsymbol{rMatr}$ einer Rotation gilt:

```
rMatr = rotMatrZ( rz ) * rotMatrY( ry ) * rotMatrX( rx )
(wobei rotMatrX, rotMatrY, rotMatrZ die 3x3-Matrizen um die X-, Y- und Z-Achse sind)
```

4x4-Matrix -> interne Darstellung Das Überführen einer in Matrix-Darstellung ***hom*** vorliegenden Transformation in die interne Darstellung erfolgt nach folgendem Prinzip:

1. Zerteile die Transformation in einen linearen Teil ***l*** und einen nicht-linearen Teil (die Translation) ***nl*** Wenn ***hom*** als 4x4-Matrix angegeben ist

   ```
   hom = ( rMatr11 rMatr12 rMatr13 sx )
         ( rMatr21 rMatr22 rMatr23 sy )
         ( rMatr31 rMatr32 rMatr33 sz )
         ( 0       0       0       1  )
   ```

   so gilt:

   ```
   l =   ( rMatr11 rMatr12 rMatr13 )
         ( rMatr21 rMatr22 rMatr23 )
         ( rMatr31 rMatr32 rMatr33 )

   nl = ( sx, sy, sz )
   ```

2. Zerteilen des linearen Teils in Rotation ***r*** und Matrix ***m***. finde ein ***r*** so, dass l = r * m, wobei ***r*** eine pure Rotation ist:

   ```
   falls l eine pure Rotation ( det(l) == 1 ):
       m = id
       rMatr = l
   ansonsten ist l eine roto-Inversion ( det(l) == -1 ):
       m = Punktspiegelung am Ursprung
       rMatr = l * m
   ```

   Das heißt ***m*** ist entweder eine Matrix für die Punktspiegelung am Ursprung, oder die Einheitsmatrix.

3. aus der 3x3-Matrix ***rMatr***, berechne die 3 Eulerwinkel rx, ry, rz so, dass die Komposition von deren Rotationsmatrizen r entspricht:

   rMatr = RotMatrZ(rz) * RotMatrY(ry) * RotMatrX(rx)

Bemerkung: Tatsächlich ist diese Zerlegung nicht eindeutig. Um mit dieser Implementierung **Kompositionseindeutigkeit** zu erreichen, ist eine eindeutige Darstellung aber essenziell. Dies kann erreicht werden, indem man sich für jede

Rotation $\boldsymbol{r}$ auf genau einen Repräsentanten in Euler-Darstellung festlegt. Unsere Implementierung stützt sich auf die folgende Arbeit von Gregory G. Slabaugh:[5]

Darin wird eine Methode vorgestellt, um aus einer Rotationsmatrix für $\boldsymbol{r}$ eine *eindeutige* Darstellung von $\boldsymbol{r}$ als Euler-Winkel zu berechnen.

interne Darstellung -> 4x4-Matrix: Das Überführen der internen Darstellung in die 4x4-Matrix ist vergleichsweise einfach, und erfolgt nach folgender Methode:

Sei $\boldsymbol{t}$ die interne Darstellung einer Transformation t: Euler-Winkel r = (rx, ry, rz) Translation s = (sx, sy, sz) Spiegelungs-Matrix oder EinheitsMatrix m

Dann berechne:

```
rMatr = RotMatrZ( rz ) * RotMatrY( ry ) * RotMatrX( rx )

hom = ( rMatr11 rMatr12 rMatr13 sx )
      ( rMatr21 rMatr22 rMatr23 sy )
      ( rMatr31 rMatr32 rMatr33 sz )
      ( 0       0       0       1  )
```

Implementierung der Kompositions-Operation Entscheidend für die korrekte Implementierung Kompositionsoperation ist das Vorliegen beider Operationen in der oben konstruierten internen Darstellung.

```
seien 2 Transformationen t1 und t2 gegeben in der internen Darstellung
    t1:
        Euler-Winkel (rx1, ry1, rz1)
        Translation (sx1, sy1, sz1)
        Spiegelungs-Matrix oder EinheitsMatrix m1
    t2:
        Euler-Winkel (rx2, ry2, rz2)
        Translation (sx2, sy2, sz2)
        Spiegelungs-Matrix oder EinheitsMatrix m2

berechne:

    t':
        Euler-Winkel r' = (rx1+rx2, ry1+ry2, rz1+rz2)
        Translation s' = (sx1+rx2, sy1+sy2, sz1+sz2)
        m = m1 * m2
```

[5] "Computing euler angles from a rotation matrix", http://www.soi.city.ac.uk/~sbbh653/publications/euler.pdf.

```
Dann ist deren Komposition t in der internen Darstellung:

    t:
        Euler-Winkel r' = rundeWinkel ( rx mod 2*pi, ry mod 2*pi, rz mod 2*pi )
        Translation s' = rundeTrans (sx1+rx2, sy1+sy2, sz1+sz2)
        m = m1 * m2

mit

    rundeWinkel (x, y, z) = komponentenweises Runden auf ggTRot
    rundeTrans (x, y, z) = komponentenweises Runden auf ggTTrans

ggTRot = 1/12
ggTRot = 1/12
```

Das heißt die Vektoren können Komponentenweise addiert werden, und die Spiegelungsmatrizen werden multipliziert.

Um die Ungenauigkeit der Fließkomma-Rechnung auszugleichen, werden nachträglich die Komponenten so gerundet, dass sie ein Vielfaches des ggT der Rotation bzw. Translation einer Raumgruppe sind. Für die Euler-Winkel muss davor noch Komponentenweise modulo 2xpi gerechnet werden, um eine eindeutige Darstellung für Rotationen zu garantieren.

Kachelung des Raumes

Voronoi Tesselierung

Für die Berechnung der 3 dimensionalen Voronoi Zellen wurde von uns das Programm QHull gewählt. QHull ist ein in C geschriebenes Programm zum Berechnen von unter anderem Delaunay Triangulationen, konvexen Hüllen und Voronoi Diagrammen in n-Dimensionen.

Die Ausgabe von Qhull ist eine Menge von indexierten Vertices. Dazu erhält man eine Menge von Mengen welche die Indexe der einzelnen Voronoi Zellen enthalten. Also [[Indices der 1. Vornoi Zelle], [Indices der 2. Vornoi Zelle] ...]

Berechnung der konvexen Hülle mit QuickHull3D

Da die Ausgabe von Qhull eine Punktmenge liefert, müssen zum darstellen der Zellwände, nun die konvexe Hülle dieser berechnet werden. Unsere Finale Lösung ist die Open Source Java Bibliothek QuickHull3D. Die nun berechneten Zellen werden in dem Objekt "Immutable Mesh" verpackt und zum Darstellen weitergeschickt.

Probleme/Herausforderungen

Die erste Herrausforderung war eine geeignete Lösung zum berechnen des Voronoi Diagrammes zu finden. Erfolglos waren wir zu beginnt auf der suche nach einer Java Bibliothek, die dieser Aufgabe gewachsen ist. Unsere engere Auswahl viel nach einiger Recherche auf PolyMake und Qhull. Polymake wurde schließlich aufgrund von Dependencies zu alten Pearl Versionen ausgesiebt. Qhull überzeugte uns da es gut dokumentiert und immer noch aktiv betreut wird. Dazu kam noch das z.B Matlab und Mathematica Qhull integriet haben, was wir als Indikator für Zuverlässigkeit und Korrektheit empfanden. Damit war unsere Entscheidung für die Voronoi Berechnung gefallen.

Für die Konvexe Hülle kam anfangs auch QHull bei uns auch zum Einsatz. Was sich zum Ende des Projektes, nach gründlicher Analyse, als schwerwiegender Flaschenhals heraus stellte. Die vielen Aufrufe des Programm für jede einzelne Zelle produzierten einen starken Overhead. Daraufhin wurde als alternative die Java Bibliothek QuickHull3d eingeführt. Eine Untersuchung durch einen Profiler ergab einen Performance Schub von ca. Faktor 100.

Zusammenfassung

Wir sind mit der finalen Lösung der Berechnung zufrieden. Da wir für beide Teilprobleme gute Lösungen gefunden haben. Selbst eine Javaimplementierung

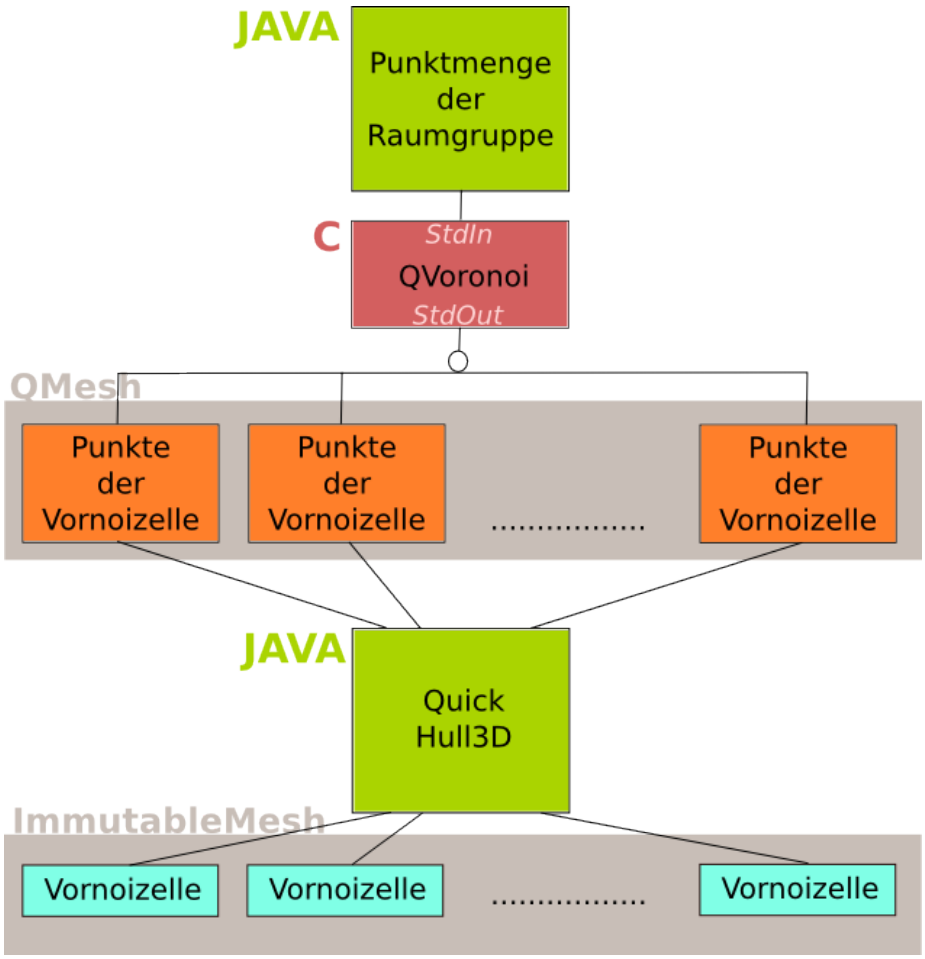


Figure 2: Schema

der Voronoiberechnung würde nur einen geringen Geschwindigkeitsvorteil bringen da diese nur einmalig beim wechseln der Raumgruppe aufgerufen wird

Färbungsalgorithmus für äquivalente Zellwände

Die Anforderung an den Algorithmus bestand darin alle gleichen Facetten(Zellwände) mit einer eindeutigen Farbe zu versehen. Der Farbbestimmung wurde mit Hilfe des folgenden Algorithmus implementiert:

- Sei **X** die Menge der Eckpunkte einer Facette
- Sei **H** ein Hashtable mit Schlüssel Farben zuweist
- Berechne den Abstand von jedem Punkt zu jedem anderen Punkt
- summiere diese auf und speichere das Ergebnis in **Z**
- Prüfe Hashtable auch Eintrag **Z**
- Falls existent => gebe Farbe zurück
- Falls nicht existent => erzeuge neuen Eintrag für **Z** und generiere eine Zufallsfarbe die noch nicht existiert

Algorithmus zum Filtern entarteter Zellen

Die Anforderung an den Algorithmus bestand darin, Zellen welche am Rande der von QVoronoi berechneten Kachelung auszufiltern, da es bei diesen zu Verzerrungen kommt. Wir gehen davon aus, dass die im Zentrum liegende Zelle korrekt ist. Diese wir als Referenz für die anderen Zellen benutzt. In Pseudo Code:

- Sei **R** die Punktmenge der Zelle welche den Zentroid (Geometrischer Schwerpunkt) der kompletten Punktmenge enthält
- Sei **A** die Menge als korrekt angenommenen Zellen
- füge **R** zu **A** hinzu
- Bilde den Zentroid von **R**
- Summiere den Abstand der Punkt von R zu seine Zentroid auf
- Speichere die Summe in **Z**
- Bilde nun von jeder anderen Zelle Zentroid
- Speichere für jede Zelle **x** den Abstand seiner Punkte zu seinem Zentroid in **Z_x**
- Für alle **Z_x** gleich **Z** => füge Zelle **x** zu **A** hinzu
- geben **A** zurück

Anmerkung : Der Algorithmus gibt keine perfekte Aussage darüber ob Zellen gleich sind. Aber im Rahmen unsere Anwendung lieferte er gute Ergebnisse.

Visualisierung

Für das Rendering der 3-Dimensionalen Voronoi-Zellen wird die OpenGL Bibliothek Jzy3d[6] verwendet. Jzy3d ist eine quelloffene Java Bibliothek, die speziell auf die Visualisierung wissenschaftlicher Daten ausgerichtet ist. Die Architektur der Benutzerschnittstelle wurde nach dem Model-View-Controller (kurz MVC) Muster entworfen. Im Model sind alle Aktivitäten gekapselt, die zur Berechnung der Punktmenge der ausgewählten Raumgruppe notwendig sind. Auf Grundlage dieser Punktmenge wird im Controller die Voronoi-Tesselierung durchgeführt. Darüber hinaus sind im Controller Statusinformationen enthalten, die einzelne Aspekte der Visualisierung beeinflussen. Sobald sich diese Statusinformationen ändern, benachrichtigt der Controller den View darüber, dass die Ansicht und die darin enthaltenen geometrischen Objekte erneuert werden müssen. Nachfolgend wird eine Übersicht über die Klassen der Visualisierung gegeben. Um eine Ausgabe zu erzeugen, muss der Nutzer eine Raumgruppe auswählen und einen Ausgangspunkt sowie die Gittergröße festlegen. Die Gittergröße legt in diesem Zusammenahng den Begrenzungsrahmen fest, in dem die Punkte liegen müssen, die durch die Tranformationen der Raumgruppe entstehen.

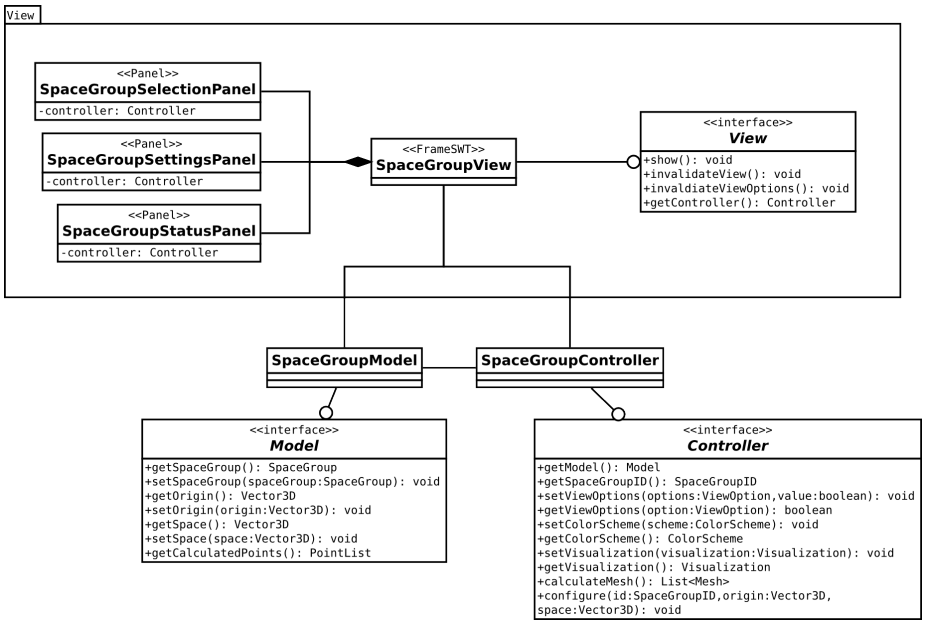


Figure 3: mvc

[6] "Jzy3d - scientific 3d plotting", http://http://jzy3d.org.

Bedienkonzept

Die Benutzeroberfläche ist so gestaltet, dass alle relevanten Informationen durch den Nutzer sofort erfasst werden können. Hierbei wurde versucht die sichtbaren Elemente hinsichtlich des logischen Zusammenhangs zu unterteilen. Um die Nutzerinterkation während der Berechnung einer Raumgruppe nicht zu beeinträchtigen, erfolgt die Berechnung asynchron. Das hat den Vorteil, dass die Oberfläche weiterhin bedienbar bleibt. Der grundlegende Aufbau der Benutzeroberfläche wird in der nachfolgenden Abbildung dargestellt.

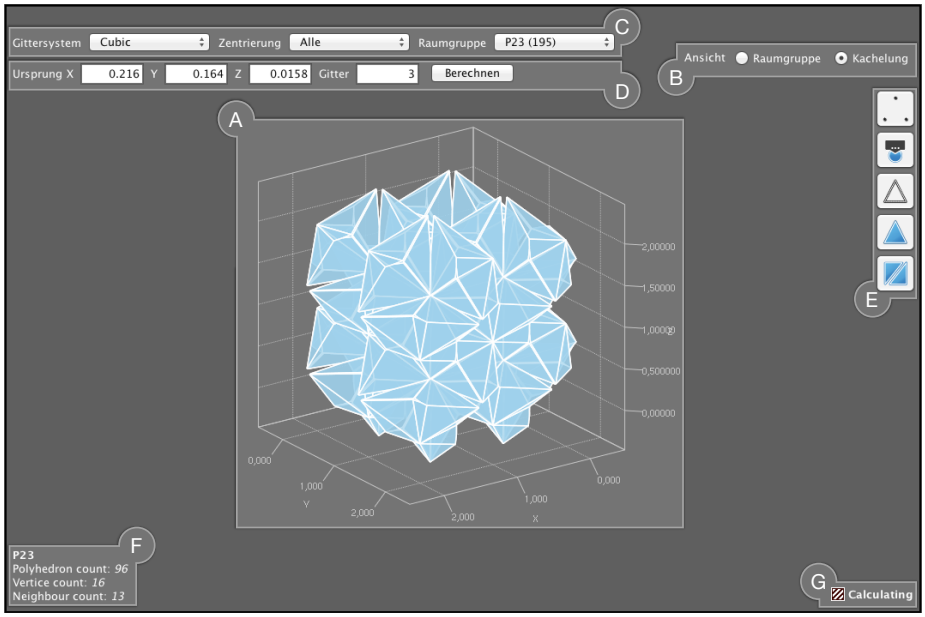


Figure 4: GUI

Die Bedeutung der einzelnen Punkte lautet wie folgt:

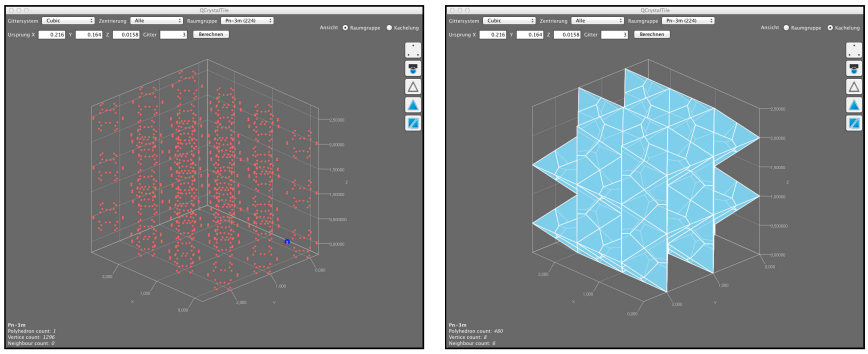
A. Gittermodell B. Art der Darstellung C. Auswahl der Raumgruppe D. Festlegung des Ausgangspunktes und der Gittergröße E. Diverse Ansichtsoptionen F. Statistik G. Aktivitätsanzeige für die Berechnung

Zusätzlich zu den sichtbaren Elementen auf der Oberfläche stehen weitere Einstellungsmöglichkeiten über ein Kontextmenü auf dem Gittermodell zur Verfügung. Alle Einstellungen, die durch den Nutzer vorgenommen wurden, werden in einem Nutzerprofil gespeichert und beim Neustart der Applikation wiederhergestellt. Die Interaktion mit dem Gittermodell erfolgt über die Maus. So kann der gesamte Inhalt gedreht werden oder durch einen Doppelklick im Ansichtsfenster automatisch rotiert werden. Um den dargestellten Ausschnitt zu vergrößern bzw. verkleinern, kann über das Mausrad der Zoom verändert werden. Ein Klick auf

eine Voronoi-Zelle bewirkt, dass diese Zelle ausgeblendet wird, um das Innere eines Kristalls bzw. Bereiche sichtbar zu machen, die vorher verdeckt wurden.

Ansichtsoptionen

Um die Auswertung der Visualisierung zu unterstützen, stehen verschiedene Optionen zur Verfügung, um einzelne Aspekte andersartig darzustellen. Eine erste Option besteht darin die Darstellung zwischen Raumgruppe und Kachelung zu wechseln. In der Darstellungsart Raumgruppe wird die berechnete Punktmenge der ausgewählten Raumgruppe als Punktwolke ausgegeben. Der blaue Punkt im Gittermodell repräsentiert den Ausgangspunkt, der für die Transformationen herangezogen wird. In dieser Visualisierung wird noch keine Voronoi-Tesselierung vorgenommen. Die Voronoi-Zellen werden erst in der Darstellung Kachelung berechnet. Die beiden Darstellungsarten sind in der folgenden Abbildung gegenübergestellt.

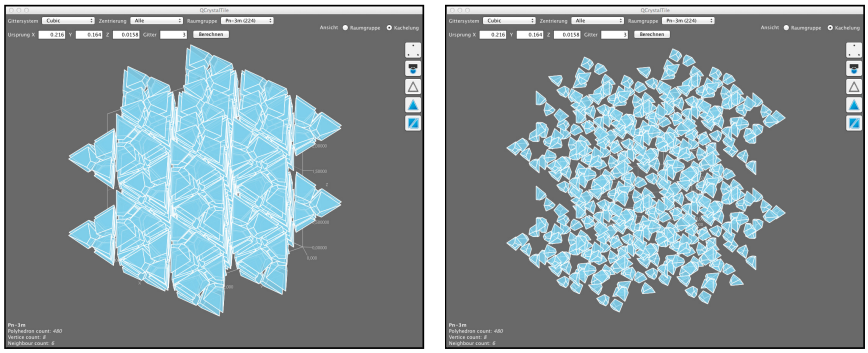


Ansicht der Punktmenge Ansicht der Kachelung

Figure 5: View

Wie bereits erwähnt hat der Nutzer die Möglichkeit einzelne Zellen auszublenden, um verdeckte Bereiche sichtbar zu machen. Eine weitere Möglichkeit, um die inneren Zellen zu enthüllen, besteht darin einen Abstand zwischen die Zellen einzufügen. Um das zu erreichen wird von jeder Zelle der Abstand zum Mittelpunkt des Gittermodells erhöht, somit entsteht der Eindruck, dass der Kristall explodiert. Nachfolgende Abbildung zeigt einen Kristall mit zwei unterschiedlichen Abstandsfaktoren.

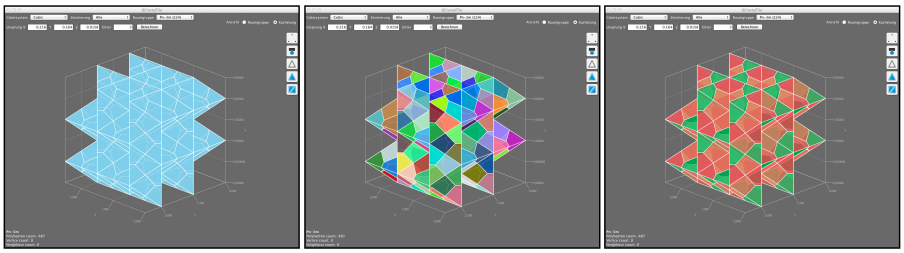
Die Zellen werden standardmäßig in einer einheitlichen Farbe dargestellt. Um die visuelle Abgrenzung der Zellen voneinander zu erhöhen, stehen weitere Farbschemen zur Verfügung. Das erste Farbschema Nach Zelle weist jeder Zelle eine eigene Farbe zu. Auf diese Weise kann die Form der Zellen besser erfasst werden. Allerdings wird nicht deutlich, wo sich die Zellen berrühren bzw. welche Kontaktflächen es gibt. Aus diesem Grund gibt es das Farbschema Nach Facette.



Abstand von 1,5x Abstaqnd von 3,5x

Figure 6: Explode

Hierbei werden den Facetten, die hinsichtlich Form und Fläche gleich sind, die selbe Farbe zugewiesen (siehe dazu Kapitel “Färbungsalgorithmus für äquivalent Zellwände”). Diese Darstellungsweise trägt ebenfalls dazu bei, zu erkennen aus wie vielen verschiedenen Facetten eine Zelle besteht. In der nachfolgenden Abbildung sind die verfügbaren Farbschemen gegenüber gestellt.



Einheitlich Nach Zelle Nach Facette

Zusammenfassung

Die Entscheidung Jzy3d zum Rendering der 3D Objekte zu verwenden, war im Nachhinein die richtige Wahl, da die Lernkurve im Umgang mit der Bibliothek relativ flach verlief und das Abstraktionsniveau so angemessen ist, das keine tiefgreifenden OpenGL Kenntnisse notwendig waren. Der einzige Nachteil bestand darin, dass die Dokumentation weitesgehend unvollständig ist. Aus diesem Grund waren die aufkommenden Probleme nicht immer einfach zu lösen. Ein solches Problem trat beispielsweise beim Entfernen von geometrischen Objekten aus der 3D Szene auf. Das passiert in dem Moment, wenn durch eine erneute Voronoi-Tesselierung massenweise Polygone entfernt werden müssen, die ungültig geworden sind. Der vorgeschriebene Weg zum Entfernen von Objekten führte zu einer starken Beeinträchtigung der Performance, was zur Folge hatte, dass die

Benutzeroberfläche zeitweise nicht reagiert hat. Insofern war es notwendig sich mit dem Quelltext von Jzy3d näher auseinander zu setzen, um eine entsprechende Lösung zu entwickeln. Ein weiteres besteht darin, dass Jzy3d bzw. die zugrundeligende Bibliothek JOGL[7] nicht mit allen Linux Distributionen kompatibel ist.

[7] "JOGL - java binding for the openGL aPI", http://jogamp.org/jogl/www/.