Belegarbeit Teil I
zur Veranstaltung Computergrafik und Visualisierung II

Günther, Emanuel s76954 - Leopold, Michael s76904

SoSe 2019

1 Aufgabenstellung

1. **Vektor2D und Vektor3D** [10 Punkte]
   Erstellen Sie die zwei Klassen `Vektor2D` und `Vektor3D`, die entsprechend 2D- oder 3D-Vektoren repräsentieren können. Die Koordinaten werden jeweils durch `double` repräsentiert. Weiterhin sollen sinnvolle Hilfsmethoden angeboten werden. Es sind mindestens die folgenden Funktionen (nachweislich) testgetrieben zu implementieren: `setPosition`, `isNullVector`, `add`, `sub`, `mult`, `div`, `isEqual`, `isNotEqual`, `length` und `normalize`. Dokumentieren Sie Ihre Entwicklung mittels Test-Driven-Development in geeigneter Weise.

2. **Lineare Algebra** [20 Punkte]
   Schreiben Sie eine Klasse `LineareAlgebra`, die folgende Methoden (für 2D und 3D) zur Verfügung stellt: `add`, `sub`, `mult`, `div`, `isEqual`, `isNotEqual`, `length`, `normalize`, `euklDistance`, `manhattanDistance`, `crossProduct`, `dotProduct`, `cosEquation`, `sinEquation`, `angleRad`, `angleDegree`, `radToDegree`, `degreeToRad`, `determinante`, `abs` und `show`. Auch hier soll die Entwicklung testgetrieben erfolgen.

2 Lösung

Die vollständigen Lösungen wurden per Email abgegeben. Hier ein Auszug der relevanten Abschnitte zu den Klassen `Vektor2D`, `Vektor3D` und `LineareAlgebra`.

Die Lösung wurde in `Eclipse 2019-03` unter Zuhilfenahme von `JUnit 5` entwickelt und getestet.

2.1 Klasse DRechnung

Um die Überprüfungen bei Berechnungen mit dem primitiven Datentyp `double` zu bündeln, wurden die Berechnungen von Summen, Differenzen, Produkten und Quotienten in die Klasse `DRechnung` ausgelagert. Dort sind sie als statische Methoden verfügbar. Weiterhin sind die größte und die kleinste mit dem Datentyp `double` darstellbare Zahl als Konstanten `MIN` und `MAX` vorhanden.

Beispielhaft für die testgetriebene Entwicklung soll die Methode `add()` vorgestellt werden. Der folgende Test überprüft eine einfache Berechnung einer Summe.

```
1  @Test
2  public void addDRTest() throws Exception {
3      double a = 5;
4      double b = 8;
5      assert (DRechnung.add(a, b) == 13);
6  }
```

Die daraufhin entwickelte Implentierung sieht wiefolgt aus:

```
1  public static double add(double a, double b) {
2      return a + b;
3  }
```

Als nächstes wird eine Überprüfung auf double Overflow hinzugefügt.

```
1  @Test
2  public void addOverflowDRTest() {
3      double a = DRechnung.MAX;
4      double b = 8;
```

```
    Assertions.assertThrows(Exception.class, () -> {
        DRechnung.add(a, b);
    });
}
```

Es folgt die Implementierung:

```
public static double add(double a, double b) throws Exception {
    if ((a > 0) && (b > MAX - a)) {
        throw new Exception("Double Overflow");
    }
    return a + b;
}
```

Zuletzt wird auf double Unterflows getestet.

```
@Test
public void addUnderflowDRTest() {
    double a = DRechnung.MIN;
    double b = -5;
    Assertions.assertThrows(Exception.class, () -> {
        DRechnung.add(a, b);
    });
}
```

Die finale Implementierung sieht wiefolgt aus:

```
public static double add(double a, double b) throws Exception {
    if ((a > 0) && (b > MAX - a)) {
        throw new Exception("Double Overflow");
    }
    if ((a < 0) && (b < MIN - a)) {
        throw new Exception("Double Underflow");
    }
    return a + b;
}
```

2.2 Klasse Vektor3D

Um den Zugriff auf die Variablen zu vereinfachen, wurden hier auf die get- und set-Methoden verzichtet:

```
public class Vektor3D {
    public double x, y, z;
    ...
}
```

Verschachtelte Konstrukturen wurden angelegt, um Redundanz zu vermeiden:

```
public Vektor3D() {
    this(0, 0, 0);
}

public Vektor3D(double x, double y, double z) {
    this.x = x;
    this.y = y;
    this.z = z;
}

public Vektor3D(Vektor3D src) {
    this(src.x, src.y, src.z);
}
```

2.2.1 Ausgewählte Methode: `length()`

Die Länge oder Magnitude eines Vektors $\vec{p}$, symbolisiert durch $\|\circ\|$, ergibt sich aus

$$length(\vec{p}) = \|\vec{p}\| = \sqrt{p_1^2 + p_2^2 + \ldots + p_n^2}$$

und liefert einen Skalar. Dem Längenmaß eines Vektors liegt der euklidische Abstand zu Grunde. Eine grafische Darstellung der Länge eines Vektors findet sich in der Abbildung 1.

Ein erster Test für die Berechnung der Länge eines Vektors ist nachfolgend zu finden:

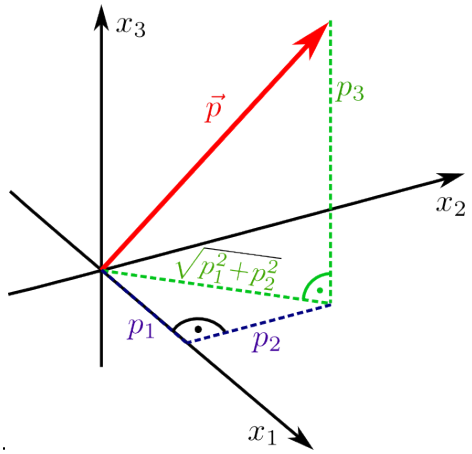

Abbildung 1: Die Länge eines dreidimensionalen Vektors.

```
@Test
public void lengthCalculation3DTest() {
   Vektor3D v = new Vektor3D(3, 4, 0);
   double l = v.length();
   assert (l == 5);
}
```

Die dazugehörige Implementierung sieht wiefolgt aus:

```
public double length() {
   return Math.sqrt(x*x + y*y + z*z);
}
```

Bei der Berechnung von Quadratzahlen und deren Summierung können schnell zu double Overflows auftreten. Der folgende Test führt eine Überprüfung dessen durch:

```
@Test
public void lengthOverflow3DTest() {
   Vektor3D v = new Vektor3D(DRechnung.MAX, 3, 4);
   Assertions.assertThrows (Exception.class, () -> {
      v.length();
   });
}
```

Die Implementierung der Overflow-Überprüfung ist jedoch nicht besonders schwer. Denn für die Berechnungen der Methode `length()` können die Methoden aus der Klasse `DRechnung` genutzt werden, die schon auf double Overflows überprüfen. Der Quellcode dafür sieht wiefolgt aus:

```
@Test
public double length() throws Exception {
   double x2 = DRechnung.mult(this.x, this.x);
   double y2 = DRechnung.mult(this.y, this.y);
   double z2 = DRechnung.mult(this.z, this.z);
   double x2y2 = DRechnung.add(x2, y2);
   return Math.sqrt(DRechnung.add(x2y2, z2));
}
```

2.3 Klasse Vektor2D

Auch in der Klasse `Vektor2D` wurden Vereinfachungen für ein einfacheres Arbeiten vorgenommen. Sie sind äquivalent zu den Vereinfachungen bei der Klasse `Vektor3D`.

2.3.1 Ausgewählte Methode: `normalize()`

Die Methode `normalize()` ist eine Auswahl unter vielen, die zeigt, dass wir unseren Code Style stets klein und Ressourceneffizeint halten. Minimalistisch und Laufzeiteffizient indem möglichst wenige Aufrufe und Bedingungen auftreten.

Die Normierung eines Vektors lässt sich wiefolgt beschreiben:

$$norm(\vec{p}) = \begin{pmatrix} p_x/length(\vec{p}) \\ p_y/length(\vec{p}) \end{pmatrix}$$

Ein erster Test zur Normierung eines zweidimensionalen Vektors sieht wiefolgt aus:

```
@Test
public void normalizeCalculation2DTest() {
    Vektor2D v = new Vektor2D(3, 4);
    v.normalize();
    assert(v.x == 3/5.);
    assert(v.y == 4/5.);
}
```

So sieht die erste Implementierung des Codes aus:

```
public void normalize() {
    this.x = DRechnung.div(this.x, length());
    this.y = DRechnung.div(this.y, length());
}
```

Ein wichtiger Test bei der Normierung von Vektoren ist, dass bei einer Länge von `length = 0` keine Division durch Null erfolgt. Der Test dafür sieht wiefolgt aus:

```
@Test
public void normalizeLengthNull2DTest() {
    Vektor2D v = new Vektor2D(0, 0);
    v.normalize();
    assert(v.x == 0);
    assert(v.y == 0);
}
```

Die Lösung dafür entspricht im Quellcode einer einfachen Überprüfung auf Null:

```
public void normalize() throws Exception {
    double len = length();
    if (len == 0) {
        return;
    }
    this.x = DRechnung.div(this.x, len);
    this.y = DRechnung.div(this.y, len);
}
```

2.4 Lineare Algebra

2.4.1 Ausgewählte Methode: `abs()` für Vektor3D

Diese Methode erhält als Parameter ein Objekt der Klasse `Vektor3D` und gibt ihn so zurück, dass negative Komponenten als positive dargestellt werden. Positive Komponenten bleiben unberührt.

Der Test für die Funktionalität dieser Methode sieht wiefolgt aus:

```
@Test
public void absCalc3DPosTest() {
   Vektor3D v1 = new Vektor3D(4, -2, 2);
   Vektor3D v2 = LineareAlgebra.abs(v1);
   assert (v2.x == 4.0);
   assert (v2.y == 2.0);
   assert (v2.y == 2.0);
}
```

Die dazugehörige Implementierung steht nachfolgend:

```
public static Vektor3D abs(Vektor3D vsrc) {
   Vektor3D v = new Vektor3D(vsrc);
   if (v.x < 0)
      v.x = -v.x;
   if (v.y < 0)
      v.y = -v.y;
   if (v.z < 0)
      v.z = -v.z;
   return v;
}
```

2.4.2 Ausgewählte Methode: `crossProduct()` für Vektor3D

Die Methode `crossProduct()` für dreidimensionale Vektoren bildet den orthogonalen Vektor $\vec{n}$ zu den beiden gegebenen $\vec{p}$ und $\vec{q}$ und gibt diesen zurück. Die Berechnungsvorschrift dafür lautet:

$$\vec{n} = \vec{p}x\vec{q} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \times \begin{pmatrix} q_x \\ q_y \\ q_z \end{pmatrix} = \begin{pmatrix} p_y q_z - p_z q_y \\ p_z q_x - p_x q_z \\ p_x q_y - p_y q_x \end{pmatrix}$$

Am besten lässt sich diese mathematische Formel mit der 'Rechten-Hand-Regel' illustrieren [2].

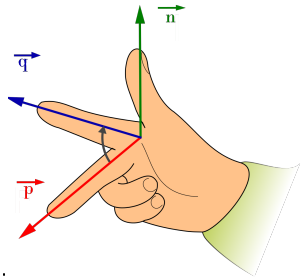

Abbildung 2: Die 'Rechte-Hand-Regel'.

Ein erster Test für das Kreuzprodukt findet sich nachfolgend:

```
@Test
public void crossProd3DLACalcTest() {
   Vektor3D v1 = new Vektor3D(1, 0, 0);
   Vektor3D v2 = new Vektor3D(0, 1, 0);
   Vektor3D v3 = LineareAlgebra.crossProduct(v1, v2);
   assert (v3.x == 0);
   assert (v3.y == 0);
   assert (v3.z == 1);
}
```

Die daraufhin entwickelte Implementierung sieht wiefolgt aus:

```
public static Vektor3D crossProduct(Vektor3D v1, Vektor3D v2) {
   double x = DRechnung.sub(DRechnung.mult(v1.y, v2.z), DRechnung.mult(v1.z, v2.y));
   double y = DRechnung.sub(DRechnung.mult(v1.z, v2.x), DRechnung.mult(v1.x, v2.z));
   double z = DRechnung.sub(DRechnung.mult(v1.x, v2.y), DRechnung.mult(v1.y, v2.x));
   return new Vektor3D(x, y, z);
}
```

Literatur

[1] Block-Berlitz M.: "*Warum sich der Dino furchtbar erschreckte - Lehrbuch zu Beleuchtung und Rendering mit Java, LWJGL, OpenGL und GLSL*", vividus Wissenschaftsverlag, 2019 (to appear :))