# Computergrafik und Visualisierung II - Beleg Teil I

Wiebke Rochler, Christopher Praas, Anna Krauß

5. Mai 2017

## Inhaltsverzeichnis

# 1 Einleitung

Dies ist die Dokumentation zum Quellcode für den Beleg im Fach Computergrafik und Visualisierung II bei Prof. Dr. Marco Block-Berlitz. Im ersten Teil des Belegs war es Aufgabe drei Klassen in Java zu implementieren. Bei der Umsetzung war darauf zu achten, dass Methoden erstellt werden, die in der Lage sind mögliche Fehler aufspüren und entsprechend zu reagieren. Die verschiedenen Klassen samt Methoden wurden in einem Package realisiert. In einem zweiten Package sind alle Tests zu den jeweiligen Methoden implementiert.

# 2 Erläuterung der verwendeten Klassen

## 2.1 Vektor2D

Die Klasse Vektor2D repräsentiert 2D-Vektoren. Die Koordinaten werden jeweils durch double-Werte dargestellt. Zunächst besitzt die Klasse drei verschiedene Konstruktoren, durch die es möglich ist auf verschiedene Art und Weisen Objekte der Klasse Vektor2D zu erstellen. Des Weiteren beinhaltet die Klasse die Grundrechenarten Addition, Subtraktion, Multiplikation und Division. Darüber hinaus werden Methoden angeboten, mithilfe derer Vektoren verglichen, normiert und an eine bestimmte Position gesetzt werden können. Außerdem kann die Vektorlänge berechnet und geprüft werden ob es sich um einen Nullvektor handelt. Alle Methoden wurden testgetrieben entwickelt, weshalb sie bestimmte Fehler erkennen und behandeln können. Wenn ein solcher Fehlerfall eintritt, wird eine Exception geworfen um dies kenntlich zu machen. Im Folgenden wird auf die einzelnen Operationen und die eventuell auftretenden Fehler eingegangen.

### 2.1.1 Methoden

**Addition:** Bei der Addition bestünde die Möglichkeit, dass es bei zu großen bzw. zu kleinen Zahlen zu einem Überlauf im positiven oder negativen Bereich kommen könnte. Dies lässt sich durch folgende Überlegungen ermitteln:

```
public void add(Vektor2D b) throws Exception
   {
      if((x<0 && b.x<0) || (x>0 && b.x>0) || (y<0 && b.y<0) || (y>0 && b.y>0))
         if((Math.abs(x)+Math.abs(b.x) >= Double.MAX_VALUE) ||
             (Math.abs(y)+Math.abs(b.y) >= Double.MAX_VALUE))
            throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
               beachten!");
```

Es wird also getestet ob die Koordinaten des aufrufenden und des übergebenen Vektors beide größer oder kleiner Null sind. Wenn das der Fall ist, wird aus diesen der Betrag gebildet und dann geprüft ob deren Addition größer/gleich dem maximalen Wert des darstellbaren Bereichs ist. Ist dies der Fall, wird eine Exception geworfen und der Nutzer über den Fehler informiert. (Mit der Bildung des Betrags der Koordinaten genügt der Test auf den Maximalwert von Double.)

```
      if((x>0 && b.x<0) || (x<0 && b.x>0) || (y>0 && b.y<0) || (y<0 && b.y>0))
         if((Math.abs(x)-Math.abs(b.x) >= Double.MAX_VALUE) ||
             (Math.abs(y)-Math.abs(b.y) >= Double.MAX_VALUE))
            throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
               beachten!");
   }
```

Des Weiteren bestünde die Möglichkeit einen sehr großen und einen kleinen Wert zu addieren, was ebenfalls zu einem Überlauf führen könnte. Daher wird getestet ob die einzelnen Koordinaten des aufrufenden und des übergebenen Vektoren größer und/oder

kleiner Null sind. Wenn das der Fall ist, wird aus diesen der Betrag gebildet und dann geprüft ob deren Subtraktion größer/gleich dem maximalen Wert des darstellbaren Bereichs ist. Ist dies der Fall, wird eine Exception geworfen und der Nutzer über den Fehler informiert.

**Subtraktion:** Prinzipiell können hier die gleichen Fehler auftreten wie bei der Addition. Es muss lediglich bei der Prüfung auf die Vorzeichen geachtet werden. Aufgrund des Bildens der Beträge in der Überprüfung auf Überlauf von der Addition können wir die gleiche Überprüfung auch für die Subtraktion verwenden.

**Multiplikation:** Bei der Multiplikation wird ein Vektor mit einer Doublezahl multipliziert. Dabei sollte auch dort darauf geachtet werden, dass kein Überlauf zustande kommt. Das verhindern wir, indem wir auch hier darauf testen ob die Operation einen größeren Wert erzeugen würde, als durch Double darstellbar. Auch hier werfen wir im Fehlerfall eine Exception und geben eine Fehlermeldung zurück.
*Programmcode zur Verdeutlichung:*

```
public void mult(double b) throws Exception
   {
      if((Math.abs(x)*Math.abs(b) >= Double.MAX_VALUE) ||
          (Math.abs(y)+Math.abs(b) >= Double.MAX_VALUE))
         throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
             beachten!");
      x*=b;
      y*=b;
   }
```

**Division:** Bei der Division muss abgesehen von dem Test auf positiven und negativen Überlauf auch getestet werden, ob der Divisor Null ist, da Division durch Null nicht zulässig ist. Wurde dies durch den Nutzer nicht beachtet, wird eine Exception geworfen und darauf hingewiesen. Für den Test auf Überlauf muss überprüft werden ob der Divisor zwischen Null und Eins liegt, da das Ergebnis in dem Fall dann größer ist als der Dividend bzw. im Falle b==1 gleich groß. Also wird auch hier der Betrag der Koordinaten gebildet und die Division auf Überlauf getestet. Im Fehlerfall erfolgt auch hier eine Exception.
*Programmcode zur Verdeutlichung:*

```
   public void div(double b) throws Exception
   {
      if(b==0)
         throw new Exception("Division durch 0 ist nicht zulässig!");

      if((b<1 && b > 0) || b==1)
         if((Math.abs(x)/Math.abs(b) >= Double.MAX_VALUE) ||
             (Math.abs(y)/Math.abs(b) >= Double.MAX_VALUE))
            throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
```

```
            beachten!");

      x/=b;
      y/=b;
   }
```

**setPosition():** Diese Funktion bietet die Möglichkeit einem Vektor durch Übergabe von x- und y-Koordinate eine neue Position zuzuweisen. Eventuell könnten diese übergebenen Werte größer sein als der double-Maximalwert, was wir durch vorheriges Prüfen ausgeschlossen haben.
*Programmcode zur Verdeutlichung:*

```
   public void setPosition(double x1, double y1) throws Exception
   {
      if((Math.abs(x1)>=Double.MAX_VALUE) || (Math.abs(y1)>=Double.MAX_VALUE))
         throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
             beachten!");

      x=x1;
      y=y1;
   }
```

**isNullVector():** In der Methode wird überprüft, ob der aufrufende Vektor ein Nullvektor ist. Dabei war nur die Korrektheit der Funktion zu prüfen.

**isEqual():** In isEqual() wird überprüft, ob der aufrufende und der übergebene Vektor den gleichen Inhalt haben. Dabei war nur die Korrektheit der Funktion zu prüfen. (Vergleich Inhalt und nicht die Objektreferenzen)

**isNotEqual():** Hierbei handelt es sich um die Umkehrfunktion von isEqual(), bei welcher wir gleichermaßen vorgegangen sind.

**length():** Die Methode length() berechnet die Länge des aufrufenden Vektors.
Mathematische Bildungsvorschrift für die Länge eines Vektors der Dimension Zwei:

$$||\vec{v}|| = \sqrt{x^2 + y^2}$$

Hier war darauf zu achten, dass das Quadrat der x- und y-Koordinaten keinen Überlauf verursacht, was durch eine Abfrage und im Fehlerfall geworfene Exception vermieden wird.

**normalize():** Auch bei der Normierung des aufrufenden Vektors wird das Quadrat der x- und y-Koordinaten auf vermeintlichen Überlauf geprüft, was durch eine Abfrage und im Fehlerfall geworfene Exception vermieden wird.

Die Normierung wird erreicht, indem der Vektor durch seine Länge geteilt wird:

$$\frac{1}{||\vec{v}||} \cdot \vec{v}$$

Hier bietet es sich an die bereits vorhandenen Funktionenen mult() und length() aufzurufen.
*Programmcode zur Verdeutlichung:*

```
public void normalize() throws Exception
{
   if(((x*x) >= Double.MAX_VALUE) || ((y*y) >= Double.MAX_VALUE))
   {
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   }
   mult(1/length());
   // double tmplaenge=Math.sqrt((x*x)+(y*y));
   // div(tmplaenge);
}
```

## 2.2 Vektor3D

Die Klasse Vektor3D repräsentiert 3D-Vektoren. Der einzige Unterschied zur Klasse Vektor2D liegt bei der Erweiterung um die z-Koordinate. Die Koordinaten werden jeweils durch double-Werte dargestellt. Zunächst besitzt die Klasse drei verschiedene Konstruktoren, durch die es möglich ist auf verschiedene Art und Weisen Objekte der Klasse Vektor3D zu erstellen. Des Weiteren beinhaltet die Klasse die Grundrechenarten Addition, Subtraktion, Multiplikation und Division. Darüber hinaus werden Methoden angeboten, mithilfe derer Vektoren verglichen, normiert und an eine bestimmte Position gesetzt werden können. Außerdem kann die Vektorlänge berechnet und geprüft werden ob es sich um einen Nullvektor handelt. Alle Methoden wurden testgetrieben entwickelt, weshalb sie bestimmte Fehler erkennen und behandeln können. Wenn ein solcher Fehlerfall eintritt, wird eine Exception geworfen um dies kenntlich zu machen. Im Folgenden wird auf die einzelnen Operationen und die eventuell auftretenden Fehler eingegangen.

### 2.2.1 Methoden

**Addition:** Bei der Addition bestünde die Möglichkeit, dass es bei zu großen bzw. zu kleinen Zahlen zu einem Überlauf im positiven oder negativen Bereich kommen könnte. Dies lässt sich durch folgende Überlegungen ermitteln:

```
public void add(Vektor3D b) throws Exception
{
   if((x<0 && b.x<0) || (x>0 && b.x>0) || (y<0 && b.y<0) || (y>0 && b.y>0)
       || (z<0 && b.z<0) || (z>0 && b.z>0))
```

```
        if((Math.abs(x)+Math.abs(b.x) >= Double.MAX_VALUE) ||
            (Math.abs(y)+Math.abs(b.y) >= Double.MAX_VALUE) ||
            (Math.abs(z)+Math.abs(b.z) >= Double.MAX_VALUE))
           throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
              beachten!");
```

Es wird also getestet ob die Koordinaten des aufrufenden und des übergebenen Vektors beide größer oder kleiner Null sind. Wenn das der Fall ist, wird aus diesen der Betrag gebildet und dann geprüft ob deren Addition größer/gleich dem maximalen Wert des darstellbaren Bereichs ist. Ist dies der Fall, wird eine Exception geworfen und der Nutzer über den Fehler informiert. (Mit der Bildung des Betrags der Koordinaten genügt der Test auf den Maximalwert von Double.)

```
     if((x>0 && b.x<0) || (x<0 && b.x>0) || (y>0 && b.y<0) || (y<0 && b.y>0)
         || (z>0 && b.z<0) || (z<0 && b.z>0))
        if((Math.abs(x)-Math.abs(b.x) >= Double.MAX_VALUE) ||
            (Math.abs(y)-Math.abs(b.y) >= Double.MAX_VALUE) ||
            (Math.abs(z)-Math.abs(b.z) >= Double.MAX_VALUE))
           throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
              beachten!");
     x+=b.x;
     y+=b.y;
     z+=b.z;
  }
```

Des Weiteren bestünde die Möglichkeit einen sehr großen und einen kleinen Wert zu addieren, was ebenfalls zu einem Überlauf führen könnte. Daher wird getestet ob die einzelnen Koordinaten des aufrufenden und des übergebenen Vektoren größer und/oder kleiner Null sind. Wenn das der Fall ist, wird aus diesen der Betrag gebildet und dann geprüft ob deren Subtraktion größer/gleich dem maximalen Wert des darstellbaren Bereichs ist. Ist dies der Fall, wird eine Exception geworfen und der Nutzer über den Fehler informiert.

**Subtraktion:** Prinzipiell können hier die gleichen Fehler auftreten wie bei der Addition. Es muss lediglich bei der Prüfung auf die Vorzeichen geachtet werden. Aufgrund des Bildens der Beträge in der Überprüfung auf Überlauf von der Addition können wir die gleiche Überprüfung auch für die Subtraktion verwenden.

**Multiplikation:** Bei der Multiplikation wird ein Vektor mit einer Doublezahl multipliziert. Dabei sollte auch dort darauf geachtet werden, dass kein Überlauf zustande kommt. Das verhindern wir, indem wir auch hier darauf testen ob die Operation einen größeren Wert erzeugen würde, als durch Double darstellbar. Auch hier werfen wir im Fehlerfall eine Exception und geben eine Fehlermeldung zurück.
*Programmcode zur Verdeutlichung:*

```
  public void mult(double b) throws Exception
```

```
{
   if((Math.abs(x)*Math.abs(b) >= Double.MAX_VALUE) ||
       (Math.abs(y)+Math.abs(b) >= Double.MAX_VALUE) ||
       (Math.abs(z)+Math.abs(b) >= Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   x*=b;
   y*=b;
   z*=b;
}
```

**Division:** Bei der Division muss abgesehen von dem Test auf positiven und negativen Überlauf auch getestet werden, ob der Divisor Null ist, da Division durch Null nicht zulässig ist. Wurde dies durch den Nutzer nicht beachtet, wird eine Exception geworfen und darauf hingewiesen. Für den Test auf Überlauf muss überprüft werden ob der Divisor zwischen Null und Eins liegt, da das Ergebnis in dem Fall dann größer ist als der Dividend bzw. im Falle b==1 gleich groß. Also wird auch hier der Betrag der Koordinaten gebildet und die Division auf Überlauf getestet. Im Fehlerfall erfolgt auch hier eine Exception.
*Programmcode zur Verdeutlichung:*

```
public void div(double b) throws Exception
{
   if(b==0)
      throw new Exception("Division durch 0 ist nicht zulässig!");

   if((b<1 && b > 0) || b==1)
      if((Math.abs(x)/Math.abs(b) >= Double.MAX_VALUE) ||
          (Math.abs(y)/Math.abs(b) >= Double.MAX_VALUE) ||
          (Math.abs(z)/Math.abs(b) >= Double.MAX_VALUE))
         throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
             beachten!");

   x/=b;
   y/=b;
   z/=b;
}
```

**setPosition():** Diese Funktion bietet die Möglichkeit einem Vektor durch Übergabe von x-, y- und z-Koordinate eine neue Position zuzuweisen. Eventuell könnten diese übergebenen Werte größer sein als der double-Maximalwert, was wir durch vorheriges Prüfen ausgeschlossen haben.
*Programmcode zur Verdeutlichung:*

```
public void setPosition(double x1, double y1, double z1) throws Exception
{
   if((Math.abs(x1)>=Double.MAX_VALUE) || (Math.abs(y1)>=Double.MAX_VALUE)
```

```
        || (Math.abs(z1)>=Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
         beachten!");
   x=x1;
   y=y1;
   z=z1;
}
```

**isNullVector():** In der Methode wird überprüft, ob der aufrufende Vektor ein Nullvektor ist. Dabei war nur die Korrektheit der Funktion zu prüfen.

**isEqual():** In isEqual() wird überprüft, ob der aufrufende und der übergebene Vektor den gleichen Inhalt haben. Dabei war nur die Korrektheit der Funktion zu prüfen. (Vergleich Inhalt und nicht die Objektreferenzen)

**isNotEqual():** Hierbei handelt es sich um die Umkehrfunktion von isEqual(), bei welcher wir gleichermaßen vorgegangen sind.

**length():** Die Methode length() berechnet die Länge des aufrufenden Vektors.
Mathematische Bildungsvorschrift für die Länge eines Vektors der Dimension Drei:

$$||\vec{v}|| = \sqrt{x^2 + y^2 + z^2}$$

Hier war darauf zu achten, dass das Quadrat der x-, y- und z-Koordinaten keinen Überlauf verursacht, was durch eine Abfrage und im Fehlerfall geworfene Exception vermieden wird.

**normalize():** Auch bei der Normierung des aufrufenden Vektors wird das Quadrat der x-, y- und z-Koordinaten auf vermeintlichen Überlauf geprüft, was durch eine Abfrage und im Fehlerfall geworfene Exception vermieden wird.
Die Normierung wird erreicht, indem der Vektor durch seine Länge geteilt wird:

$$\frac{1}{||\vec{v}||} \cdot \vec{v}$$

Hier bietet es sich an die bereits vorhandenen Funktionenen mult() und length() aufzurufen.
*Programmcode zur Verdeutlichung:*

```
public void normalize() throws Exception
{
   if(((x*x) >= Double.MAX_VALUE) || ((y*y) >= Double.MAX_VALUE) || ((z*z)
      >= Double.MAX_VALUE))
   {
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
         beachten!");
```

```
    }
    mult(1/length());
    // double tmplaenge=Math.sqrt((x*x)+(y*y)+(z*z));
    // div(tmplaenge);
  }
```

## 2.3 LineareAlgebra

Die Klasse LineareAlgebra stellt verschiedene Methoden für 2D- und 3D-Vektoren bereit. Da diese Methoden alle statisch sind, wird ein privater Konstruktor benötigt um sicher zu stellen, dass kein Objekt von LineareAlgebra angelegt wird.

```
private LineareAlgebra(){}
```

Die verschiedenen Funktionen werden im folgenden näher erläutert. Bei den Grundrechenarten Addition, Subtraktion, Multiplikation und Division wurde zur Berechnung des Ergebnisses ein temporärer Vektor erstellt.

### 2.3.1 Methoden

**Addition mit Rückgabetyp Vektor2D:** Bei der Addition bestünde die Möglichkeit, dass es bei zu großen bzw. zu kleinen Zahlen zu einem Überlauf im positiven oder negativen Bereich kommen könnte. Dies lässt sich durch folgende Überlegungen ermitteln:

```
    public static Vektor2D add(Vektor2D a, Vektor2D b) throws Exception
  {
    if((a.x<0 && b.x<0) || (a.x>0 && b.x>0) || (a.y<0 && b.y<0) || (a.y>0 &&
        b.y>0))
      if((Math.abs(a.x)+Math.abs(b.x) >= Double.MAX_VALUE) ||
          (Math.abs(a.y)+Math.abs(b.y) >= Double.MAX_VALUE))
        throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
            beachten!");
```

Es wird also getestet ob die Koordinaten der übergebenen Vektoren beide größer oder kleiner Null sind. Wenn das der Fall ist, wird aus diesen der Betrag gebildet und dann geprüft ob deren Addition größer/gleich dem maximalen Wert des darstellbaren Bereichs ist. Ist dies der Fall, wird eine Exception geworfen und der Nutzer über den Fehler informiert. (Mit der Bildung des Betrags der Koordinaten genügt der Test auf den Maximalwert von Double.)

```
    if((a.x>0 && b.x<0) || (a.x<0 && b.x>0) || (a.y>0 && b.y<0) || (a.y<0 &&
        b.y>0)) //beide unterschiedlich
      if((Math.abs(a.x)-Math.abs(b.x) >= Double.MAX_VALUE) ||
          (Math.abs(a.y)-Math.abs(b.y) >= Double.MAX_VALUE))
        throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
            beachten!");
```

```
    Vektor2D c = new Vektor2D(a.x+b.x,a.y+b.y);
    return c;
}
```

Des Weiteren bestünde die Möglichkeit einen sehr großen und einen kleinen Wert zu addieren, was ebenfalls zu einem Überlauf führen könnte. Daher wird getestet ob die einzelnen Koordinaten der übergebenen Vektoren größer und/oder kleiner Null sind. Wenn das der Fall ist, wird aus diesen der Betrag gebildet und dann geprüft ob deren Subtraktion größer/gleich dem maximalen Wert des darstellbaren Bereichs ist. Ist dies der Fall, wird eine Exception geworfen und der Nutzer über den Fehler informiert.

**Addition mit Rückgabewert Vektor3D:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass der Rückgabetyp Vektor3D ist und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**Subtraktion mit Rückgabetyp Vektor2D:** Prinzipiell können hier die gleichen Fehler auftreten wie bei der Addition. Es muss lediglich bei der Prüfung auf die Vorzeichen geachtet werden. Aufgrund des Bildens der Beträge in der Überprüfung auf Überlauf von der Addition können wir die gleiche Überprüfung auch für die Subtraktion verwenden.

**Subtraktion mit Rückgabetyp Vektor3D:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass der Rückgabetyp Vektor3D ist und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**Multiplikation mit Rückgabteyp Vektor2D:** Bei der Multiplikation wird ein Vektor mit einer Doublezahl multipliziert. Dabei sollte auch dort darauf geachtet werden, dass kein Überlauf zustande kommt. Das verhindern wir, indem wir auch hier darauf testen ob die Operation einen größeren Wert erzeugen würde, als durch Double darstellbar. Auch hier werfen wir im Fehlerfall eine Exception und geben eine Fehlermeldung zurück.
*Programmcode zur Verdeutlichung:*

```
public static Vektor2D mult(Vektor2D a, double b) throws Exception
{
   if((Math.abs(a.x)*Math.abs(b) >= Double.MAX_VALUE) ||
       (Math.abs(a.y)+Math.abs(b) >= Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");

   Vektor2D c= new Vektor2D();
   c.x=a.x*b;
   c.y=a.y*b;
   return c;
}
```

**Multiplikation mit Rückgabetyp Vektor3D:** Der Unterschied zur vorhergehenden

Funktion besteht darin, dass der Rückgabetyp Vektor3D ist und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**Division mit Rückgabetyp Vektor2D:** Da die Division nichts anderes ist als die Umkehrfunktion der Multiplikation, können wir diese hier ganz einfach mit dem Reziproke aufrufen. Es muss abgesehen von dem Test auf positiven und negativen Überlauf auch getestet werden, ob der Divisor Null ist, da Division durch Null nicht zulässig ist. Wurde dies durch den Nutzer nicht beachtet, wird eine Exception geworfen und darauf hingewiesen. Für den Test auf Überlauf muss überprüft werden ob der Divisor zwischen Null und Eins liegt, da das Ergebnis in dem Fall dann größer ist als der Dividend bzw. im Falle b==1 gleich groß. Also wird auch hier der Betrag der Koordinaten gebildet und die Division auf Überlauf getestet. Im Fehlerfall erfolgt auch hier eine Exception. *Programmcode zur Verdeutlichung:*

```
public static Vektor2D div(Vektor2D a, double b) throws Exception
{
   if(b==0)
      throw new Exception("Division durch 0 ist nicht zulässig!");

   if((b<1 && b > 0) || b==1)
      if((Math.abs(a.x)/Math.abs(b) >= Double.MAX_VALUE) ||
          (Math.abs(a.y)/Math.abs(b) >= Double.MAX_VALUE))
         throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
             beachten!");

   return (mult(a, 1/b));
}
```

**Division mit Rückgabetyp Vektor3D:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass der Rückgabetyp Vektor3D ist und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**isEqual()mit zwei übergebenen Vektor2D- oder zwei Vektor3D-Objekten:** In den Funktionen isEqual() wird überprüft, ob die übergebenen Vektoren den gleichen Inhalt haben. Dabei war nur die Korrektheit der Funktion zu prüfen. (Vergleich Inhalt und nicht die Objektreferenzen)

**isNotEqual()mit zwei übergebenen Vektor2D- oder zwei Vektor3D-Objekten:** Hierbei handelt es sich um die Umkehrfunktionen von isEqual(), bei welcher wir gleichermaßen vorgegangen sind.

**length() von Vektor2D oder Vektor3D:** Die Methoden length() berechnen die Länge des jeweils übergebenen Vektors.
Hier war darauf zu achten, dass das Quadrat der x-, y- und (im Falle von length() für

Vektor3D) z-Koordinaten keinen Überlauf verursacht, was durch eine Abfrage und im Fehlerfall geworfene Exception vermieden wird.

**normalize() mit Rückgabetyp Vektor2D:** Auch bei der Normierung des übergebenen Vektors wird das Quadrat der x-, y- Koordinate auf vermeintlichen Überlauf geprüft, was durch eine Abfrage und im Fehlerfall geworfene Exception vermieden wird.
Die Normierung wird erreicht, indem der Vektor durch seine Länge geteilt wird:

$$\frac{1}{||\vec{v}||} \cdot \vec{v}$$

Hier bietet es sich an die bereits vorhandenen Funktionenen mult() und length() aufzurufen.
*Programmcode zur Verdeutlichung:*

```
public static Vektor2D normalize(Vektor2D a) throws Exception
{
   if(((a.x*a.x) >= Double.MAX_VALUE) || ((a.y*a.y) >= Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");

    return (mult(a, 1/length(a)));
}
```

**normalize() mit Rückgabetyp Vektor3D:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass der Rückgabetyp Vektor3D ist und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**euklDistance() mit zwei übergebenen Vektor2D-Objekten:** Die euklidische Distanz für zwei 2D-Vektoren bezeichnet den Abstand dieser in der Ebene. Die Funktion wurde auf Basis von folgender mathematischen Formel programmiert:

$$euklDistance(p_1, p_2) = \sqrt{(p_2.x - p_1.x)^2 + (p_2.y - p_1.y)^2}$$

Dabei war darauf zu achten, dass die Differenz der beiden Vektorkoordinaten im Quadrat auf vermeintlichen Überlauf geprüft wird und dies im Fehlerfall mit einer Exception behandelt wird. *Programmcode zur Verdeutlichung:*

```
public static double euklDistance(Vektor2D a, Vektor2D b) throws Exception
{
   if((Math.pow(a.x-b.x,2)>=Double.MAX_VALUE) ||
       (Math.pow(a.y-b.y,2)>=Double.MAX_VALUE) ||
       (Math.sqrt(Math.pow(a.x-b.x,2)+Math.pow(a.y-b.y,2))>=Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");

   double diffX=Math.pow(a.x-b.x,2);
```

```
    double diffY=Math.pow(a.y-b.y,2);
    double dist=Math.sqrt(diffX+diffY);
    return dist;
}
```

**euklDistance() mit zwei übergebenen Vektor3D-Objekten:** Die euklidische Distanz für zwei 3D-Vektoren bezeichnet den Abstand dieser im Raum. Die Funktion wurde auf Basis von folgender mathematischen Formel programmiert:

$$euklDistance(p_1, p_2) = \sqrt{(p_2.x - p_1.x)^2 + (p_2.y - p_1.y)^2 + (p_2.z - p_1.z)^2}$$

Der Unterschied zur vorhergehenden Funktion besteht darin, dass zwei Vektor3D-Objekte übergeben werden und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**manhattanDistance() mit zwei übergebenen Vektor2D-Objekten:** Die Manhattan-Distanz für zwei 2D-Vektoren bezeichnet die Distanz zwischen diesen als die Summe der absoluten Differenzen ihrer Einzelkoordinaten. Die Funktion wurde auf Basis von folgender mathematischen Formel programmiert:

$$manhattanDistance(p_1, p_2) = |p_2.x - p_1.x| + |p_2.y - p_1.y|$$

Dabei war darauf zu achten, dass die Differenz der beiden Vektorkoordinaten im Betrag auf vermeintlichen Überlauf geprüft wird und dies im Fehlerfall mit einer Exception behandelt wird.
*Programmcode zur Verdeutlichung:*

```
public static double manhattanDistance(Vektor2D a, Vektor2D b) throws
    Exception
{
   if((Math.abs(b.x-a.x)>=Double.MAX_VALUE) ||
       (Math.abs(b.y-a.y)>=Double.MAX_VALUE) ||
       ((Math.abs(b.x-a.x)+Math.abs(b.y-a.y))>=Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   return (Math.abs(b.x-a.x)+ Math.abs(b.y-a.y));
}
```

**manhattanDistance() mit zwei übergebenen Vektor3D-Objekten:** Die Manhattan-Distanz für zwei 3D-Vektoren bezeichnet die Distanz zwischen diesen als die Summe der absoluten Differenzen ihrer Einzelkoordinaten. Die Funktion wurde auf Basis von folgender mathematischen Formel programmiert:

$$manhattanDistance(p_1, p_2) = |p_2.x - p_1.x| + |p_2.y - p_1.y| + |p_2.z - p_1.z|$$

Der Unterschied zur vorhergehenden Funktion besteht darin, dass zwei Vektor3D-Objekte übergeben werden und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**crossProduct() mit zwei übergebenen Vektor2D-Objekten:** Das Kreuzprodukt existiert eigentlich nur für Vektoren mit der Dimension Drei, weshalb es für 2D nicht näher betrachtet wurde. Gemeint ist wahrscheinlich die Determinante ⇒ siehe Abschnitt determinante().

**crossProduct() mit zwei übergebenen Vektor3D-Objekten:** Das Kreuzprodukt definiert einen Vektor, der senkrecht auf der aufgespannten Ebene beiden Ausgangsvektoren (der beiden übergebenen Vektoren) steht. Mathematisch ist es folgendermaßen definiert:

$$\vec{u} \times \vec{v} = \begin{pmatrix} u_2v_3 - u_3v_2 \\ u_3v_3 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{pmatrix}$$

*Programmcode zur Verdeutlichung:*

```
public static Vektor3D crossProduct(Vektor3D a, Vektor3D b) throws Exception
{
   if(((a.x*b.y)>=Double.MAX_VALUE) || ((a.x*b.z)>=Double.MAX_VALUE) ||
       ((a.z*b.y)>=Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");

   Vektor3D c= new Vektor3D();
   c.x=((a.y*b.z)-(a.z*b.y));
   c.y=((a.z*b.x)-(a.x*b.z));
   c.z=((a.x*b.y)-(a.y*b.x));
   return c;
}
```

Auch bei dieser Funktion war es wichtig auf Überlauf zu testen, welche durch die Multiplikation der Einzelkoordinaten der 3D-Vektoren hervorgerufen werden könnte. Im Fehlerfall wird auch hier eine Exception geworfen und der Nutzer erhält eine Mitteilung. Auf die Überprüfung auf gleiche Dimension der übergebenen Vektoren können wir verzichten, da wir explizit nur Objekte der Klasse Vektor3D als Übergabeparameter zulassen.

**dotProduct() mit zwei übergebenen Vektor2D-Objekten:** Diese Funktion bietet die mathematische Verknüpfung zweier Vektoren. Das Ergebnis ist ein Skalar (eine reelle Zahl). Mathematisch ist es folgendermaßen definiert:

$$\vec{u} \cdot \vec{v} = u_1 \cdot v_1 + u_2 \cdot v_2$$

Durch die testgetriebene Entwicklung reagiert auch diese Funktion auf einen Überlauf durch Multiplikation der Einzelkoordinaten und wirft in einem solchen Falle eine Exception. Auf die Überprüfung auf gleiche Dimension der übergebenen Vektoren können wir verzichten, da wir explizit nur Objekte der Klasse Vektor2D als Übergabeparameter

zulassen.
*Programmcode zur Verdeutlichung:*

```
public static double dotProduct(Vektor2D a, Vektor2D b) throws Exception
{
   if(((a.x*b.x)+(a.y*b.y))>=Double.MAX_VALUE)
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   return ((a.x*b.x)+(a.y*b.y));
}
```

**dotProduct() mit zwei übergebenen Vektor3D-Objekten:** Diese Funktion bietet die mathematische Verknüpfung zweier Vektoren. Das Ergebnis ist ein Skalar (eine reelle Zahl). Mathematisch ist es folgendermaßen definiert:

$$\vec{u} \cdot \vec{v} = u_1 \cdot v_1 + u_2 \cdot v_2 + u_3 \cdot v_3$$

Der Unterschied zur vorhergehenden Funktion besteht darin, dass zwei Vektor3D-Objekte übergeben werden und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**cosEquation() mit zwei übergebenen Vektor2D-Objekten:** cosEquation() beinhaltet die Berechnung des Cosinus des von den beiden Vektoren eingeschlossenen Winkels und ist es folgendermaßen definiert:

$$\cos \alpha = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \cdot |\vec{v}|}$$

Die Berechnung besteht also aus den bereits vorhandenen Funktionen dotProduct() und length(), welche daher von cosEquation() genutzt werden. Ein möglicher Überlauf kann daher nur bei der Multiplikation der jeweiligen Längen der Vektoren auftreten, weshalb dies zuvor abgefragt wird. Im Fehlerfall wird eine entsprechende Exception geworfen. Der Winkel $\alpha$ selbst ergibt sich dann wie im Abschnitt angleDegree() besprochen.
*Programmcode zur Verdeutlichung:*

```
public static double cosEquation(Vektor2D a, Vektor2D b) throws Exception
{
   if((dotProduct(a,b)>=Double.MAX_VALUE) ||
       ((length(a)*length(b)>=Double.MAX_VALUE)))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");

   double cos=dotProduct(a,b);
   cos/=(length(a)*length(b));
   return cos;
}
```

**cosEquation() mit zwei übergebenen Vektor3D-Objekten:** Der Unterschied zur

vorhergehenden Funktion besteht darin, dass zwei Vektor3D-Objekte übergeben werden.

**sinEquation() mit zwei übergebenen Vektor2D-Objekten:** Diese Methode berechnet den Sinus des, von den beiden Vektoren eingeschlossenen Winkels und ist es folgendermaßen definiert:

$$\sin\alpha = \frac{|\vec{u} \times \vec{v}|}{|\vec{u}| \cdot |\vec{v}|}$$

*Programmcode zur Verdeutlichung:*

```
public static double sinEquation(Vektor2D a, Vektor2D b) throws Exception
{
   if((crossProduct(a,b)>=Double.MAX_VALUE) ||
       ((length(a)*length(b)>=Double.MAX_VALUE)))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");

   double sin=crossProduct(a,b);
   sin/=(length(a)*length(b));
   return sin;

}
```

Auch in dieser Methode konnten vorhandene Methoden wiederverwendet werden, nämlich crossProduct() und length(). Ein möglicher Überlauf kann daher nur bei der Multiplikation der jeweiligen Längen der Vektoren auftreten, weshalb dies zuvor abgefragt wird. Im Fehlerfall wird eine entsprechende Exception geworfen.

**sinEquation() mit zwei übergebenen Vektor3D-Objekten:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass zwei Vektor3D-Objekte übergeben werden.

**angleRad() mit zwei übergebenen Vektor2D-Objekten:** Diese Funktion liefert den Winkel zwischen den übergebenen Vektoren im Bogenmaß. Hier haben wir uns einfach unserer Kosinusfunktion bedient, welche durch den Arkuskosinus invertiert wird. Die acos-Funktion liefert in Java immer das Ergebnis in Bogenmaß. Wir haben den Fall abgefangen, dass zu große Zahlen übergeben werden könnten.
*Programmcode zur Verdeutlichung:*

```
public static double angleRad(Vektor2D a, Vektor2D b) throws Exception
{
   if((a.x>=Double.MAX_VALUE) || (a.y>=Double.MAX_VALUE) ||
       (b.x>=Double.MAX_VALUE) || (b.y>=Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   return Math.acos(cosEquation(a, b));
}
```

**angleRad() mit zwei übergebenen Vektor3D-Objekten:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass zwei Vektor3D-Objekte übergeben werden und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**radToDegree():** radToDegree() wandelt Bogenmaß in Gradmaß um. Die Voraussetzung dazu ist diese Formel:

$$deg = \frac{360°}{2\pi} \cdot rad$$

Der mögliche Überlauf durch die Multiplikation wird dabei von der Methode abgefangen und mit einer entsprechenden Exception behandelt.
*Programmcode zur Verdeutlichung:*

```
public static double radToDegree(double a) throws Exception
{
   if(((2 * Math.PI) * a)>=Double.MAX_VALUE)
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   return ((360 / (2 * Math.PI)) * a);
}
```

**angleDegree() mit zwei übergebenen Vektor2D-Objekten:** Diese Funktion liefert den Winkel zwischen den übergebenen Vektoren im Gradmaß. Hier haben wir uns einfach unserer angleRad-Funktion bedient, welche den Winkel im Bogenmaß liefert und durch die Funktion radToDegree()in Gradmaß umgewandelt wird. Dabei haben wir den Fall abgefangen, dass zu große Zahlen übergeben werden könnten.
*Programmcode zur Verdeutlichung:*

```
public static double angleDegree(Vektor2D a, Vektor2D b) throws Exception
{
   if((a.x>=Double.MAX_VALUE) || (a.y>=Double.MAX_VALUE) ||
       (b.x>=Double.MAX_VALUE) || (b.y>=Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   return radToDegree(angleRad(a, b));
}
```

**angleDegree() mit zwei übergebenen Vektor3D-Objekten:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass zwei Vektor3D-Objekte übergeben werden und die Abfrage auf Überlauf um die z-Koordinaten erweitert wurde.

**degreeToRad():** Diese Methode wandelt Gradmaß in Bogenmaß um. Die Voraussetzung dazu ist diese Formel:

$$rad = \frac{2\pi}{360°} \cdot deg$$

Sie bildet also die Umkehrfunktion von radToDegree() und wird ebenso auf Überlauf getestet.
*Programmcode zur Verdeutlichung:*

```
public static double degreeToRad(double a) throws Exception
{
   if((((2 * Math.PI) / 360) * a)>=Double.MAX_VALUE)
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   return (((2 * Math.PI) / 360) * a);
}
```

**determinante() mit zwei übergebenen Vektor2D-Objekten:** Die Funktion berechnet, unter der Voraussetzung das zwei 2D-Vektoren übergeben wurden (welche zusammen eine 2 x 2-Matrix darstellen), die Determinante nach folgendem Schema:

$$\vec{u} \times \vec{v} = \left(u_1 v_2 - u_2 v_1\right)$$

Es ist bei 2D-Vektoren auch unter dem Namen Kreuzprodukt verbreitet, da es nichts anderes ist als das Kreuzprodukt der Dimension Drei mit der z-Koordinate auf 0 gesetzt. Daher rufen wir hier einfach die Funktion crossProduct() auf und fangen natürlich vorher wieder alle eventuellen Überlaufmöglichkeiten ab.
*Programmcode zur Verdeutlichung:*

```
public static double determinante(Vektor2D a, Vektor2D b) throws Exception
{
   if((a.x>=Double.MAX_VALUE) || (a.y>=Double.MAX_VALUE) ||
       (b.x>=Double.MAX_VALUE) || (b.y>=Double.MAX_VALUE))
      throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
          beachten!");
   return (crossProduct(a,b));
}
```

**determinante() mit drei übergebenen Vektor3D-Objekten:** Die Funktion berechnet, unter der Voraussetzung das drei 3D-Vektoren übergeben wurden (welche zusammen eine 3 x 3-Matrix darstellen), die Determinante nach der Regel von Sarrus:

$$det(A) = u_{11}u_{22}u_{33} + u_{12}u_{23}u_{31} + u_{13}u_{21}u_{32} - u_{13}u_{22}u_{31} - u_{11}u_{23}u_{32} - u_{12}u_{21}u_{33}$$

*Programmcode zur Verdeutlichung:*

```
public static double determinante(Vektor3D a, Vektor3D b, Vektor3D c)
    throws Exception
{
   if((a.x*b.y*c.z + b.x*c.y*a.z + c.x*a.y*b.z - c.x*b.y*a.z - a.x*c.y*b.z -
       b.x*a.y*c.z)>=Double.MAX_VALUE)
```

```
        throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
            beachten!");

    return (a.x*b.y*c.z + b.x*c.y*a.z + c.x*a.y*b.z - c.x*b.y*a.z -
        a.x*c.y*b.z - b.x*a.y*c.z);
}
```

Hier wird geprüft ob das Ergebnis der Berechnung einen Überlauf erzeugen würde und dies im Fehlerfall durch eine Exception abgefangen.

**abs() mit Rückgabetyp Vektor2D:** Der Funktion wird ein 2D-Vektor übergeben und von diesem jeweils x- und y-Koordinate in den Betrag gesetzt. Die Einzelkoordinaten werden vor der Betragsbildung auf Überlauf getestet.
*Programmcode zur Verdeutlichung:*

```
public static Vektor2D abs(Vektor2D a) throws Exception
{
    if((a.x>=Double.MAX_VALUE) || (a.y>=Double.MAX_VALUE))
        throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
            beachten!");
    Vektor2D c= new Vektor2D(Math.abs(a.x),Math.abs(a.y));
    return c;
}
```

**abs() mit Rückgabetyp Vektor3D:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass ein Vektor3D-Objekt übergeben wird und die Abfrage auf Überlauf um die z-Koordinate erweitert wurde.

**show() mit übergebenen Vektor2D-Objekt:** show() ist dafür zuständig die Einzelkoordinaten des übergebenen Vektors auszugeben. Vor der Ausgabe werden diese auf Überlauf getestet.
*Programmcode zur Verdeutlichung:*

```
public static void show(Vektor2D a) throws Exception
{
    if((a.x>=Double.MAX_VALUE) || (a.y>=Double.MAX_VALUE))
        throw new Exception("Ueberlauf! Bitte den maximalen Wertebereich
            beachten!");
    System.out.println("Die x-Koordinate: " + a.x + "\n" + "Die y-Koordinate:
        " + a.y);
}
```

**show() mit übergebenen Vektor3D-Objekt:** Der Unterschied zur vorhergehenden Funktion besteht darin, dass ein Vektor3D-Objekt übergeben wird und die Abfrage auf Überlauf um die z-Koordinate erweitert wurde.

# 3 Testverfahren

Für die testgetriebene Entwicklung wurde JUnit verwendet. Bei den Tests haben wir vor allem auf Überlauf getestet, da dies bei den meisten der Methoden eine mögliche Fehlerquelle darstellt. Dazu wurden Rechnungen mit dem maximalen und minimalen Zahlenwerten des Typs Double durchgeführt, um sicher zu stellen, dass die Methoden mit Überlauf korrekt umgeht.
Bei der Überprüfung auf Korrektheit der Rechnungen wurde mit verschiedenen Zahlenwerten im Doublebereich gearbeitet um zu gewährleisten, dass die Methoden mit verschiedenen Werten fehlerfrei arbeiten.