



# DER NIESEN DAR- GESTELLT IN JAVA

Arbeiten mit JavaFX und  
der Methode Ray Casting

Maturaarbeit, be-  
treut durch Pascal  
Schuppli

Silvan Spiess

Gymnasium  
Biel-Seeland

Matura 2019  
Klasse 19c

# INHALTSVERZEICHNIS

<b>1</b>	<b>EINFÜHRUNG.....</b>	<b>3</b>
1.1	WARUM DIESES THEMA.....	3
1.2	ZIEL DER ARBEIT.....	4
1.3	WARUM ETWAS MACHEN, DAS ES SCHON GIBT?.....	4
1.4	WELCHE PROGRAMMIERSPRACHE PASST ZU MIR? .....	4
1.5	AUSGANGSSITUATION .....	5
1.6	DIE METHODE «RAY-CASTING» .....	5
1.6.1	RAY CASTING VS RAY TRACING.....	6
<b>2</b>	<b>VORSTUFEN DES 3D-MODELLS.....</b>	<b>7</b>
2.1	DIE HÖHENDATEN IN DER KONSOLE AUSGEBEN .....	7
2.2	DIE 2D SEITENANSICHT DES MODELLS .....	8
2.3	DER WÜRFEL ALS ERSTES 3D OBJEKT.....	9
2.3.1	ROTATION UND TRANSLATION DES 3D-MODELLS .....	10
2.4	MIT MESH ZU EINEM 3-DIMENSIONALEN OBJEKT .....	11
<b>3</b>	<b>DAS GELÄNDE ALS TERRAIN MESH.....</b>	<b>13</b>
3.1	ARBEITEN MIT TexCOORDS .....	16
<b>4</b>	<b>DER RAY CASTER.....</b>	<b>18</b>
4.1	RAYCASTER, DIE MAIN KLASSE.....	20
<b>5</b>	<b>FAZIT ZUR ARBEIT .....</b>	<b>24</b>
5.1	FAZIT ZUM TERRAINMESH .....	24
5.2	FAZIT ZUM RAY CASTER .....	25
<b>6</b>	<b>ANHANG.....</b>	<b>27</b>
6.1	QUELLENVERZEICHNIS .....	27
6.2	BILDVERZEICHNIS .....	28

<b>6.3</b>	<b>DIE PROGRAMME AUSFÜHREN .....</b>	<b>29</b>
<b>7</b>	<b>REDLICHKEITSERKLÄRUNG .....</b>	<b>30</b>
<b>8</b>	<b>ANHANG.....</b>	<b>31</b>
<b>8.1</b>	<b>ZEICHNER .....</b>	<b>31</b>
<b>8.1.1</b>	<b>HAUPTPROGRAMMZEICHNER.....</b>	<b>31</b>
<b>8.1.2</b>	<b>ZEICHNER .....</b>	<b>32</b>
<b>8.1.3</b>	<b>DERLESER .....</b>	<b>34</b>
<b>8.1.4</b>	<b>AUSGEBER .....</b>	<b>35</b>
<b>8.2</b>	<b>BOXZEICHNER .....</b>	<b>36</b>
<b>8.2.1</b>	<b>HAUPTPROGRAMMBOX.....</b>	<b>36</b>
<b>8.2.2</b>	<b>BOXZEICHNER .....</b>	<b>36</b>
<b>8.3</b>	<b>PYRAMIDMESH .....</b>	<b>38</b>
<b>8.3.1</b>	<b>HAUPTPROGRAMMPM .....</b>	<b>38</b>
<b>8.3.2</b>	<b>PYRAMIDMESH .....</b>	<b>39</b>
<b>8.4</b>	<b>TERRAINMESH .....</b>	<b>41</b>
<b>8.4.1</b>	<b>HAUPTPROGRAMMTERRAINMESH.....</b>	<b>41</b>
<b>8.4.2</b>	<b>DERLESERCASTER .....</b>	<b>42</b>
<b>8.4.3</b>	<b>MYMAP.....</b>	<b>44</b>
<b>8.4.4</b>	<b>TERRAINMESHNIESEN .....</b>	<b>46</b>
<b>8.5</b>	<b>CASTER.....</b>	<b>50</b>
<b>8.5.1</b>	<b>HAUPTPROGRAMMCASTER.....</b>	<b>50</b>
<b>8.5.2</b>	<b>RAYCASTER.....</b>	<b>51</b>

# 1 EINFÜHRUNG

---

## 1.1 WARUM DIESES THEMA

Da ich als Hobby Orientierungslauf betreibe, habe ich auch in Betracht gezogen, etwas mit Karten zu machen. Ein weiterer Punkt, welcher mich reizte, war das Programmieren, denn im fakultativen Informatikunterricht hatte ich ein erstes Spiel in Python programmiert und das hat mich wirklich gepackt. Bei diesem Spiel handelte es sich um ein 2-dimensionales Autorennen (Sicht von oben), wobei es beim Programmieren auch schon ein bisschen Mathematik benötigte, damit die Autos korrekt um die Kurve fahren. Der Weg, wie ich zu meiner eigentlichen Maturarbeit kam, führt jedoch über mehrere Destinationen. Angefangen hat alles, als wir (ich und meine Familie) in den USA in den Ferien waren. Unter anderem besuchten wir die Stadt San Francisco an der Westküste des Landes. Vieles an dieser Stadt hatte mich sehr beeindruckt, doch was mir am meisten imponierte, war, wie uneben die Stadt war. In San Francisco gibt es unglaublich steile Hänge, was für mich eine neue Erfahrung war. Dort kam mir dann die Idee, die Stadt in einem 3D Modell darzustellen und in eine App einzubauen. Als Funktion sollte man in der App verschiedene Routen von Punkt «A» zu Punkt «B» erhalten, alle mit verschiedenen Eigenschaften, zum Beispiel kürzeste Distanz, kleinste Höhendifferenz zu bewältigen, das Verhältnis dieser beiden Werte kombiniert usw. Herr Schuppli musste mich dann ein bisschen abbremsen, da mein Vorhaben locker zwei Maturarbeiten füllen würde (1. Gelände modellieren, 2. App erstellen) und ich sollte mich besser für ein Projekt entscheiden. Dies leuchtete mir schnell ein und so entschied ich mich für Punkt 1, also ein Gelände zu modellieren. Ich habe mich für die Programmiersprache Java entschieden. Herr Schuppli schlug vor, dieses Modell später mit der Methode «Ray-Tracing» zu bearbeiten – mehr dazu werde ich später erläutern – was mich sehr interessierte und ich es deshalb in meine Maturarbeit einbaute. Nachdem ich die ersten Informationen bearbeitete, realisierte ich, dass San Francisco vielleicht doch

nicht die beste Option ist, da die Stadt, wenn man sie von einer gewissen Distanz betrachtet, flach wirkt, was bei meiner Maturarbeit den Effekt mindern könnte. Meine Mutter, welche aus dem Gürbetal (zwischen Bern und Thun) stammt, schlug mir dann den Gantrisch vor. Ich merkte schnell, dass der Gantrisch ein geeigneter Berg ist, um ihn 3-dimensional abzubilden, da er mit 2175 M.ü.M. eine gewisse Höhe hat, bei der man beim Betrachten des Modells eine klare Geländestruktur feststellen kann. Nachdem ich dann aber mit der eigentlichen Programmierarbeit begonnen hatte, musste ich nochmal auf ein anderes Objekt zum 3D-Darstellen ausweichen, nämlich auf den Niesen. Dieser ist noch exponierter und daher für meine Maturarbeit noch besser geeignet. Nach diesem längeren Gedankengang konnte ich schliesslich das Ziel meiner Maturarbeit definieren und mich an das Umsetzen meiner Arbeit machen.

## 1.2 ZIEL DER ARBEIT

Das zu erreichende Ziel einer Maturarbeit war es, mit der Programmiersprache Java ein 3-D Modell des Niesens zu erstellen und dieses anschliessend mit einem Ray Caster darzustellen.

## 1.3 WARUM ETWAS MACHEN, DAS ES SCHON GIBT?

Von Google oder lokalen Institutionen wie der Bau-, Verkehrs- und Energiedirektion des Kantons Bern gibt es schon spezialisierte Kartenmodelle, die mein Modell um vieles übertreffen. Doch was mich an dieser Arbeit interessiert, ist wie ein solches Programm funktioniert und was für eine Mathematik dahintersteckt.

## 1.4 WELCHE PROGRAMMIERSPRACHE PASST ZU MIR?

Als ich mit der Maturarbeit begann, hatte ich ein leichtes Vorwissen über Python und kannte einige Begriffe in Java. Dies liess meine Wahl auf eine dieser beiden Programmiersprache fallen. Was schlussendlich ausschlaggebend war, waren die Graphics Datenbanken in Java, welche ich als hilfreich für meine

Rasternetzlandschaft empfunden habe. Deshalb habe ich mich für die Programmiersprache Java entschieden.

### 1.5 AUSGANGSSITUATION

Mir war von Anfang an klar, dass ich mir zuallererst ein Grundwissen über Java und seine Begriffe aufbauen musste. Aufgrund dessen klickte ich mich erstmals auf Youtube durch diverse Java Tutorials, um mir einen Überblick über diese Programmiersprache zu verschaffen [1]. Des Weiteren gab mir mein Vater das Lehrbuch «The Java Tutorial» von Mary Campoine und Kathy Walrath, welches die Grundkonzepte von Java beschreibt.

### 1.6 DIE METHODE «RAY-CASTING»

Das sogenannte Ray-Casting stammt aus den 1960er Jahren und wurde oft verwendet, um eine 3-dimensionale Szene schnell darstellen zu können. Diese Methode war in den ersten Computerspielen sehr gebräuchlich (das bekannteste unter ihnen ist das Spiel «Wolfenstein 3D»), zumal die Computer in den Jahren 1970-1980 eine geringere Arbeitsleistung hatten als die Computer heute. Ray Casting wird oft auch als vereinfachte Version des komplexeren Ray Tracings angesehen, worauf ich jedoch im Kapitel 2.6.1 eingehen werde. In der Methode Ray Casting wird eine räumliche Anordnung von Objekten oder Räumen auf einem 2-dimensionalen Bildschirm abgebildet, man nennt dies auch Schein-3D. Dabei wird von einem Sichtpunkt durch jede einzelne Pixelspalte eines Sichtfensters (zum Beispiel der Bildschirm des Computers) ein Lichtstrahl geschossen und solange verfolgt, bis er auf eine Oberfläche trifft. Wenn der Lichtstrahl auf eine Fläche gestossen ist, wird die Länge des Lichtstrahls berechnet. Mit der Länge des Strahls kann ermittelt werden, wie gross ein Objekt dargestellt werden muss.

Der Vorteil dieser Methode ist, dass von einer gewissen Szene oder Landschaft nur der Teil gezeichnet werden muss, welcher wirklich im Bildschirm sichtbar ist. Trifft

ein Strahl auf kein Objekt, so wird dieser nach einer festgelegten Strecke gestoppt und ein neuer Strahl durch die nächste Pixelspalte auf dem Bildschirm geschossen. Wie man in Abbildung 1 sehen kann, ist die Höhe der Wände abhängig zur Länge des Strahls. Dies löst den 3D Effekt aus, da weit entfernt Objekte kleiner erscheinen.

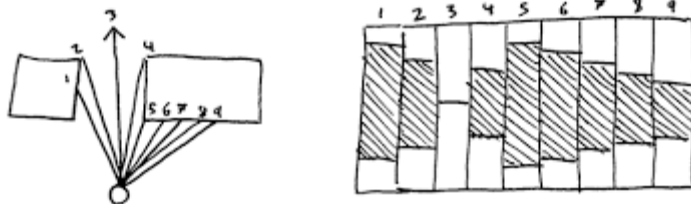


Abbildung 1 Schematische Darstellung des Ray Casting [2]

### 1.6.1 Ray Casting vs Ray Tracing

Ray Tracing ist im Gegensatz zu Ray Casting eine genauere Rendermethode, eine Methode, um eine 3-dimensionale Szene auf dem Bildschirm abbilden zu können. Dies erreicht man, indem man mit Ray Tracing Schatten und Spiegelungen darstellen kann. Anders als beim Ray Casting wird beim Ray Tracing durch jeden Pixel des Bildschirms ein Strahl gesendet, welcher an beliebig vielen Objekten in der Szene abprallt, bis er schliesslich zur Lichtquelle trifft und im Bildschirm gezeichnet werden kann. Beim Ray Tracing braucht es eine viel genauere Vektorgeometrie, wes-

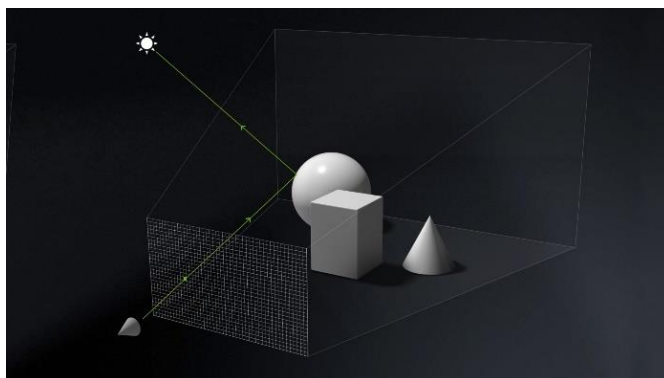


Abbildung 2 Beispiel einer mit Ray Tracing dargestellten Szene [3]

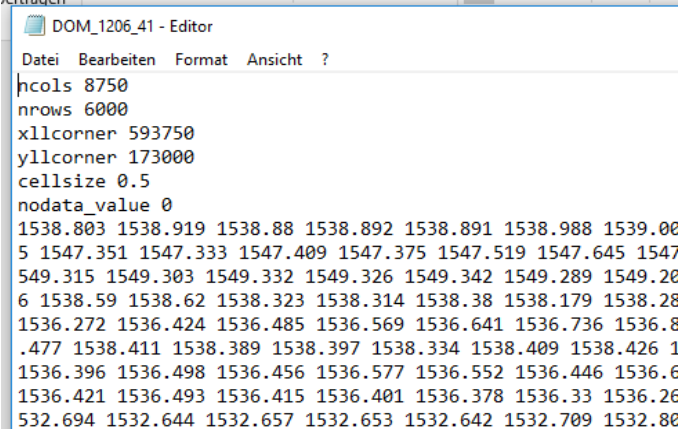
halb diese Methode anspruchsvoller und zeitaufwändiger ist, als das Ray Casting, welches ich in meiner Arbeit verwende.

## 2 VORSTUFEN DES 3D-MODELLS

---

### 2.1 DIE HÖHENDATEN IN DER KONSOLE AUSGEBEN

Zu Beginn meiner Arbeit benötigte ich exakte Höhendaten. Die Höhendaten bekam ich von der Website «Geoportal des Kantons Bern», wo sie frei zum Download waren [4]. Der Datensatz, welchen ich benötigte, bestand aus zwei Typen. Der erste Typ war eine «asc»



```
DOM_1206_41 - Editor
Datei Bearbeiten Format Ansicht ?
ncols 8750
nrows 6000
xllcorner 593750
yllcorner 173000
cellsize 0.5
nodata_value 0
1538.803 1538.919 1538.88 1538.892 1538.891 1538.988 1539.00
5 1547.351 1547.333 1547.409 1547.375 1547.519 1547.645 1547
549.315 1549.303 1549.332 1549.326 1549.342 1549.289 1549.20
6 1538.59 1538.62 1538.323 1538.314 1538.38 1538.179 1538.28
1536.272 1536.424 1536.485 1536.569 1536.641 1536.736 1536.8
.477 1538.411 1538.389 1538.397 1538.334 1538.409 1538.426 1
1536.396 1536.498 1536.456 1536.577 1536.552 1536.446 1536.6
1536.421 1536.493 1536.415 1536.401 1536.378 1536.33 1536.26
532.694 1532.644 1532.657 1532.653 1532.642 1532.709 1532.80
```

Abbildung 3 Datensatz der «asc»-Daten im Editor

Datei und der zweite war vom Typ «xyz». Der Name «asc» beziehungsweise ascending kommt vom englischen und heisst aufsteigend, da diese Datei nur die Höhenangaben beinhaltet, wogegen beim Typ «xyz» die Koordinaten der X-, Y- und Z-Achsen angegeben sind. Insgesamt ist die Datei gut 2,5 Gigabyte gross, wobei die von mir verwendeten «asc»-Daten eine Grösse von etwa 1,75 Gigabyte haben. Dieses Datenbündel wurde jedoch in vier gleich grosse Stücke aufgeteilt. Jedes dieser Stücke überspannt eine etwa zwölf Quadratkilometer grosse Fläche mit einem Punktrasternetz, dessen Punkte in einem Abstand von 0,5 Meter liegen. Für jedes Datenstück sind sechs allgemeine Informationen über die Datei gegeben, zum Beispiel Zeilen- und Spaltenzahl oder die genauen Koordinaten des oberen linken Punktes dieser Datei, sowie die 52'500'000 Höhenangaben zu jedem Punkt. Dieses Punktrasternetz ist in 6000 Zeilen mit jeweils 8750 Punkte auf einer Zeile angeordnet. Ich erstellte einen zweidimensionalen Array, welcher zuerst mit den Angaben abgefüllt wird und die Daten dann in der Konsole ausgegeben werden. Mein nächster Schritt war, diese Daten 2-dimensional darzustellen.



## 2.2 DIE 2D SEITENANSICHT DES MODELLS

Diese Seitenansicht diente einerseits dazu, dass ich ein Gefühl für Grafiken in Java aufbauen konnte, andererseits war es auch eine Standortbestimmung um zu sehen, wie gut ich mit Java umgehen konnte. Weil meine Daten 6000 Zeilen umfassen und die Layer dieser Seitenansicht ab circa 100 Layer nicht mehr zu unterscheiden wären, beschränkte ich mich auf

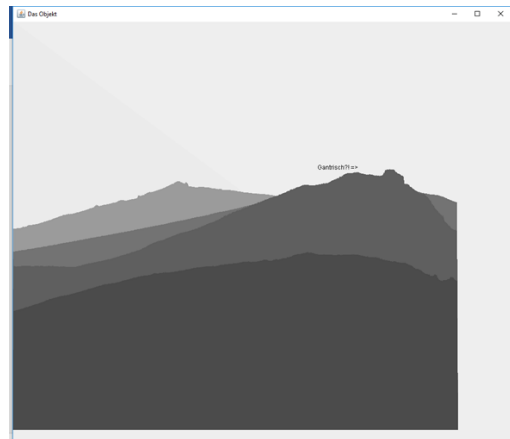


Abbildung 4 Ganttrisch mit vier sichtbaren Layern

10 Layer. Auch bei den Punkten auf einer Zeile – es stehen 8750 Punkte zur Verfügung – nahm ich nur jeden zehnten Punkt, um eine unnötig längere Rechenzeit zu vermeiden. Ich erstellte dazu zwei verschachtelte Arrays; den ersten, um die Layer zu dezimieren, und den Zweiten, um die Punkte auf den Layern zu reduzieren. Weiter generierte ich einen Farbverlauf, vom hintersten Layer (weiss) zum vordersten Layer (schwarz), um es möglichst realistisch zu gestalten. Das Rechteck-Rasternetz, welches die Daten beinhaltet, ist in vier Rechtecke unterteilt, wobei nicht ersichtlich war, in welchem dieser Rechtecke sich der Ganttrisch befindet. Durch das Darstellen der Daten konnte ich mir einen Überblick verschaffen, jedoch war der Ganttrisch nicht eindeutig zu lokalisieren. In Abbildung 4 ist eine mögliche Ausgabe des Ganttrischs zu sehen, das Gelände passt jedoch mit der wirklichen Geländestruktur nicht wirklich zusammen. Wie in Kapitel 1.1 erklärt, habe ich mich im Verlauf der Arbeit dazu entschieden, mich auf den Niesen zu fokussieren. Die nächsten Programme werde ich deshalb nicht mehr mit den Höhendaten des Ganttrisch, sondern mit denjenigen des Niesen arbeiten.

## 2.3 DER WÜRFEL ALS ERSTES 3D OBJEKT

Der nächste Schritt in Java war die Darstellung eines Würfels, in Java «Box» genannt. Die Grafikbibliothek JavaFX eignete sich gut für meine Arbeit, da sie ziemlich schlicht aufgebaut ist und man sich schnell einarbeiten kann. In JavaFX gibt es «predefined objects» wie eine Box, einen Zylinder und eine Kugel, welche sehr einfach zu generieren sind, da man praktisch nur Masse, Position und fakultativ eine Bewegung festlegen muss damit Java das Objekt generiert. Um ein Gelände darzustellen, ist es jedoch am effektivsten, wenn man die Oberfläche mit vielen Polygo-

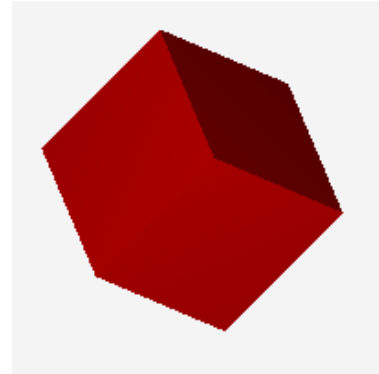


Abbildung 5 roter Würfel in JavaFX

```
package alleProgramme;

import javafx.animation.RotateTransition;

public class BoxZeichner {

    PerspectiveCamera cam = new PerspectiveCamera(false);
    BorderPane pane;
    Box box;

    //Konstruktor für Box
    public BoxZeichner()
    {
        this.box = new Box(100,100,100);
        this.pane = new BorderPane();
    }
}
```

Abbildung 6 Programmcode zum Erstellen der Box

nen (z.B. Dreiecken) übersät. Doch Polygone, sogenannte «user-defined objects», werden später behandelt. Der Scene, in welcher der Würfel generiert werden sollte, übergab ich die Masse 600 x 600 Pixel. Für mei-

nen Würfel wählte ich 100 Pixel als Höhe, Breite und Länge. Zusätzlich bestand mein Würfel aus einem sogenannten «Phong Material», weil man dem Würfel so sehr einfach eine Farbe übergeben kann, in diesem Fall Rot.

### 2.3.1 Rotation und Translation des 3D-Modells

Für die Rotation beziehungsweise die Translation eines Würfels, gibt es zwei Varianten zur Realisierung: Man kann eine Bewegung eines Objekts mit einer Animation durchführen, wobei nach dem Start des Programms keine Anpassungen mehr vorgenommen werden können. Die zweite Variante ist mithilfe von «Slidern» interaktiv nach dem Start des Programms eine Bewegung durchzuführen. Bei beiden Methoden braucht es definierte Angaben, wie an welcher Achse gedreht werden muss, oder wie lange gedreht werden soll. Die

```
Point3D p3 = new Point3D(0, 1, 0);
box.setRotationAxis(p3);

Slider SR3 = new Slider(0, 360, 180);
SR3.setShowTickLabels(true);
SR3.setShowTickMarks(true);
SR3.setMajorTickUnit(60);
SR3.setMinorTickCount(60);
SR3.valueProperty().bindBidirectional(box.rotateProperty());
pane.setBottom(SR3);
```

Abbildung 7 Programmcode eines Sliders (SR3) für eine Rotation um 360° um die Y-Achse

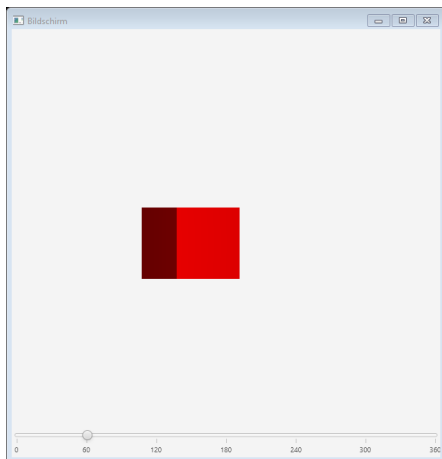


Abbildung 8 rotierter Würfel um Y-Achse

Achse konnte ich mithilfe eines «Point 3D» genau bestimmen. Ein Point 3D ist ein Punkt oder ein Vektor im Koordinatensystem. Wenn man diesem Point 3D die X-Achse zuweist, so kann man um den Point 3D rotieren, was das Ganze einfacher macht, weil der Point 3D die Koordinaten von X-, Y-, Z-Achse bezieht und man somit nicht die Namen der Achsen schreiben muss. Wie in Abbildung 8 zu sehen ist, ist der Würfel nur von der Seite zu betrachten, doch bei einer Landschaft wäre es schöner, sie auch von oben betrachten zu können. Um dies zu erreichen, habe ich den Würfel um 30°, um die X-Achse gedreht. Den Slider habe ich um denselben Wert in entgegengesetzter Richtung gedreht, um ihn nicht zu verzerren. So mit entsteht ein Würfel, welcher kubischer aussieht als in Abbildung 8.

Achse konnte ich mithilfe eines «Point 3D» genau bestimmen. Ein Point 3D ist ein Punkt oder ein Vektor im Koordinatensystem. Wenn man diesem Point 3D die X-Achse zuweist, so kann man um den Point 3D rotieren, was das Ganze einfacher macht, weil der Point 3D die Koordinaten von X-, Y-, Z-Achse bezieht und man somit nicht die Namen der Achsen

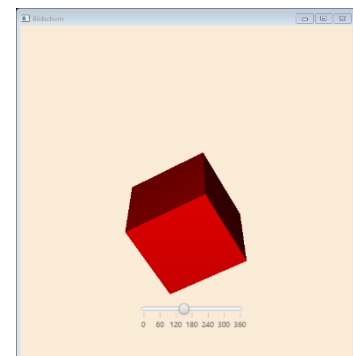


Abbildung 9 gedrehter und rotierter Würfel

## 2.4 MIT MESH ZU EINEM 3-DIMENSIONALEN OBJEKT

```
public void pyramid(Stage Bildschirm) throws Exception
{
    pyramidMesh.getTexCoords().addAll(0,0);

    pyramidMesh.getPoints().addAll(
        0, 0, 0, //P1
        0, 150, -150, //P2
        -150, 150, 0, //P3
        150, 150, 0, //P4
        0, 150, 150 //P5
    );

    pyramidMesh.getFaces().addAll(
        0,0, 2,0, 1,0, //Front Links
        0,0, 1,0, 3,0, //Front Rechts
        0,0, 3,0, 4,0, //Hinten Rechts
        0,0, 4,0, 2,0, //Hinten Links
        4,0, 2,0, 1,0, //Unten Hinten
        1,0, 3,0, 4,0 //Unten Vorne
    );
}
```

Abbildung 9 Programmcode für eine Pyramide mit Meshs

In JavaFX sind Mesh Flächen, die zum Zeichnen von Objekten mit komplexen Oberflächen-strukturen geeignet sind. So zum Beispiel auch bei einem Geländemodell, wie ich es zeichnen werde. Meshobjekte brauchen immer drei Angaben: die TexCoords, die Points und die Faces. TexCoords sind da, um die Orientierung einer Abbildung auf einem Mesh bestimmen

zu können. Bei einer ein-

fachen Pyramide ist die von geringer Bedeutung, weshalb ich die Koordinaten für den Orientierungspunkt auf 0/0 gesetzt haben, doch JavaFX braucht diese Informationen, um das Mesh erstellen zu können. Die zweite Angabe, welche JavaFX braucht, sind die Koordinaten der Eckpunkte (oben erwähnt als Points) des Objekts. Im Veranschaulichungsbeispiel sind die vier Eckpunkte der Grundfläche und der Punkt auf der Spitze der Pyramide. Für diese Punkte sind X-, Y- und Z-Koordinaten verlangt, wobei auf die Darstellung der Punkte zu achten ist, damit kein Durcheinander entsteht. Die dritte und wichtigste Angabe ist die Zuteilung der Punkte zu einer bestimmten Fläche (oben erwähnt als Faces). JavaFX erstellt die Flächen im Gegenuhrzeigersinn, was wichtig ist zu beachten, um die richtigen Punkte für die nachfolgenden Flächen zu übertragen. Jede Fläche enthält also die drei Punkte, sowie die TexCoords zu jedem Punkt, die bei meinem Beispiel Null sind. Die TexCoords werden später in dieser Arbeit genauer erklärt. Die quadratische Grundfläche wird auch in zwei Dreiecke aufgeteilt, da JavaFX mit Meshs ausschliesslich Dreiecke zeichnet. So entsteht die in Abbildung 10 (Seite 12) dargestellte Pyramide, wobei ich den Programm Code aus dem Internet übernommen und leicht angepasst habe [5].

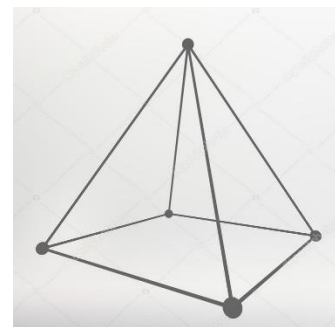


Abbildung 8 Veranschaulichen des Beispiel einer Pyramide mit Eckpunkten

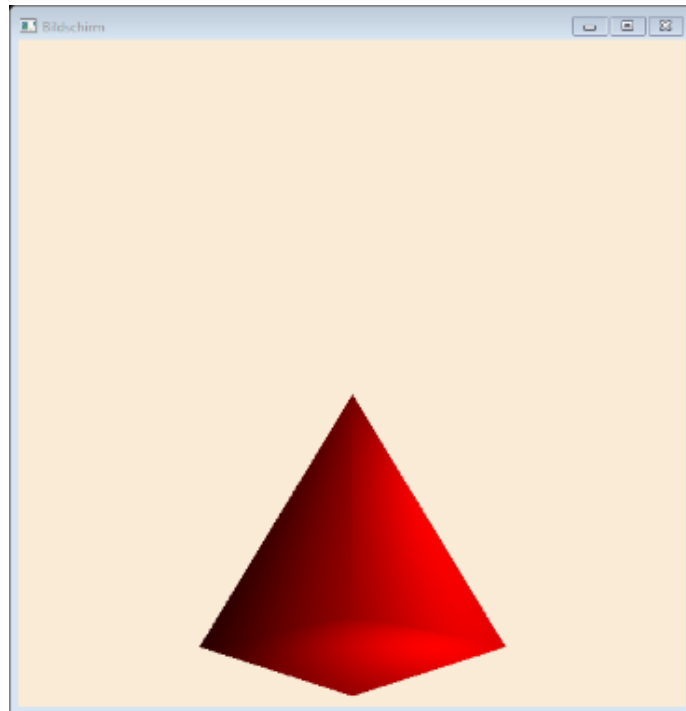


Abbildung 10 Rote Meshpyramide zum Programmcode auf Seite 11

### 3 DAS GELÄNDE ALS TERRAIN MESH

---

Nun geht es darum, das Gelände erstmals vollständig darzustellen. Dieses Gelände ist ähnlich zu programmieren wie die zuvor erwähnte Mesh Pyramide. Ein Unterschied besteht darin, dass man nun anstatt den fünf festgelegten Punkten, (bei vollem Umfang des Geländes) 52'500'000 Punkte iterieren und ebenso viele Mesh generieren muss. Dies habe ich mithilfe von

zwei «for»-Schlaufen (auch Loops genannt) umgesetzt. Die erste Schlaufe ruft jede einzelne Zeile ab und pro Zeile wird die zweite angeregt, welche auf jeder dieser Zeilen jeden Punkt abrufen. Dies kann man in der Abbildung 11 betrachten. So wird jeder Punkt auf dem Punktrasternetz ins Geländemodell eingefügt. In der inneren Schlaufe wird eine if-then-else Anweisung

```
public void MyMesh(Stage Bildschirm) throws Exception
{
    int nPoints1 = (nColumns / 100)+2;
    int yPoints1P1 [] = new int [nPoints1];
    int yPoints1P2 [] = new int [nPoints1];
    int yPoints1P3 [] = new int [nPoints1];
    int yPoints1P4 [] = new int [nPoints1];

    TriangleMesh Mesh11 = new TriangleMesh();

    //TexCoords der Dreiecke
    Mesh11.getTexCoords().addAll(0, 0);

    //1. Loop für jede Zeile
    for(int z11 = 0; z11 < (nRows / 100) -1; z11++)
    {
        //if (y11 % 100 == 0)
        System.out.println("z11= " + z11);

        //2. Loop für jede Spalte (Punkt auf Zeile)
        for(int x11 = 0; x11 < (nColumns / 100) -1; x11++)
        {
            //if(x11 % 100 == 0)
            System.out.println("x11= " + x11);
        }
    }
}
```

Abbildung 11 Programmcode der beiden Loops

durchgeführt, die nur jeder hundertste Punkt im Geländemodell darstellt. Zu Beginn der Methode wird die Länge der «nPoints» festgelegt (die Division durch 100 ist darauf zurückzuführen, dass ich auf meinem Computer nur jeden 100sten Punkt abbilden liess, um Rechenzeit zu sparen), damit die von JavaFX benötigte Länge der Arrays yPointsP1-4 bestimmt werden kann. Dies stellte lange ein Problem dar, denn zu Beginn war die Angabe, wie lange die yPointsP1-4 Arrays sein sollten, falsch, was Java als einen Fehler ansah. Dadurch musste ich dann jede einzelne Längenangabe von Arrays in meinen Programmen überprüfen, zumal durch die Dezimierung der Punkte noch jede dieser Längenangaben angepasst werden musste. Ich liess immer zwei Mesh in einem Quadrat aus vier Punkten darstellen, was am geeignetsten erschien. Dies kann man sich gut anhand einer Matrix vorstellen. Das erste Mesh im Quadrat wurde aus den Punkten  $a_{11}$ ,  $a_{22}$  und  $a_{12}$  generiert und das zweite Mesh aus

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & & & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Abbildung 12 Veranschaulichungs-  
Beispiel einer Matrix

den Punkten  $a_{11}$ ,  $a_{21}$  und  $a_{22}$ . Wiederum ist darauf zu achten, dass die Reihenfolge der Punkte im Gegenuhrzeigersinn ist.

Der nächste Schritt ist, wie schon bei der Mesh Pyramide die Punkte zu generieren und den Mesh zuzuordnen. Das

Koordinatensystem in Java ist dem 2-dimensionalen Bildschirm angepasst. Deshalb ist die Y-Achse die senkrechte Achse und die Z-Achse anstelle der X-Achse die räumliche Achse. In meinem Programm wirkt sich dies wie folgt aus: Die X- und die Z-Achse bilden die Grundfläche, während die Y-Achse ihre Werte aus den Höhendaten übernimmt. Die Y-Werte sind negativ gesetzt, da, wie in Abbildung 15 gezeigt wird, im 2-dimensionalen Bildschirm der Nullpunkt der obere linke Punkt ist, die Y-Achse am linken Bildschirmrand nach unten verläuft und das Gelände so über dem Nullpunkt gezeichnet wird.

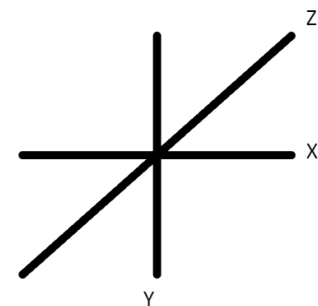


Abbildung 13 Koordinatensystem  
in Java (Beispiel)

```

a1 += 4;
a2 += 4;
a3 += 4;
b1 += 4;
b2 += 4;
b3 += 4;

yPoints1P1[x11] = (arrHeights[x11][z11]);
yPoints1P2[x11] = (arrHeights[x11 + 1][z11]);
yPoints1P3[x11] = (arrHeights[x11][z11 + 1]);
yPoints1P4[x11] = (arrHeights[x11 + 1][z11 + 1]);

//Alle Punkte eines Quadrats
Mesh11.getPoints().addAll(
    x11 * 10,          (-yPoints1P1[x11]),      z11 * 10,
    (x11 + 1) * 10,    (-yPoints1P2[x11]),      z11 * 10,
    x11 * 10,          (-yPoints1P3[x11]),      (z11 + 1) * 10,
    (x11 + 1) * 10,    (-yPoints1P4[x11]),      (z11 + 1) * 10
);

//Fläche der beiden Dreiecke in einem Quadrat
Mesh11.getFaces().addAll(
    a1,0, a2,0, a3,0,
    b1,0, b2,0, b3,0
);

```

Abbildung 14 Programmcode der Methoden «getPoints» und  
«getFaces»

Ein weiteres Problem entstand bei der Zuteilung der Punkte zu den Mesh. Denn wenn ein Mesh gezeichnet wurde, müssen die Punkte um vier erhöht werden, damit keine doppelte Nennung entsteht. Tut man dies nicht, wird einfach immer das gleiche Mesh gezeichnet, welches durch seine Grösse, dann sehr schwer zu finden ist und auch nicht das Ziel meiner

Arbeit ist. Deshalb werden zu Beginn, in Abbildung 14, die vier Punkte des jeweiligen Quadrates um 4 erhöht.

Wie oben bereits erwähnt, ist die Richtung, in welche die Punkte angeordnet sind, wesentlich, damit die Fläche korrekt gezeichnet wird. Falls man die Punkte im Uhrzeigersinn anordnete, so wird die Fläche von unten gezeichnet, da dann dort die Punkte im Gegenuhrzeigersinn angeordnet sind. Um dies zu vermeiden, gibt es zwei Lösungswege. Erstens kann man die Punkte einmal im Gegenuhrzeigersinn (a1, a2, a3) und einmal im Uhrzeigersinn (a3, a2, a1) anordnen, so werden von jeder Seite ein Mesh gezeichnet und es werden auf beiden Seiten die Fläche gezeichnet.

```
//Fläche der beiden Dreiecke in einem Quadrat
Mesh11.getFaces().addAll(
    a1,Tx1, a2,Tx2, a3,Tx3,
    b1,Tx1, b2,Tx2, b3,Tx3,
    a3,Tx3, a2,Tx2, a1,Tx1,
    b3,Tx3, b2,Tx2, b1,Tx1
);
```

```
MyMesh1 = new MeshView(Mesh11);
MyMesh1.setMaterial(mat);
MyMesh1.setCullFace(CullFace.NONE);
MyMesh1.setDrawMode(DrawMode.FILL);
MyMesh1.setVisible(true);
root.getChildren().add(MyMesh1);
```

Abbildung 15 Programmcode der Zuordnung der Punkte, sowie der CullFace Funktion

Der zweite Lösungsweg beinhaltet den Gebrauch der von Java erstellten Funktion CullFace. Mit CullFace kann man bestimmen, welche Seite der Fläche gezeichnet werden soll. CullFace.FRONT und CullFace.BACK sind für die Vorder- und Rückseite und CullFace.NONE bedeutet, dass keine Seite bevorzugt wird. In meinem Programm habe ich die CullFace auf NONE gesetzt, da ich die Punkte bereits in den beiden Uhrzeigersinnen angeordnet habe. In den Abbildungen 16 und 17 ist der Unterschied zu sehen. In Abbildung 16 ist ein Teil des Geländes unsichtbar, doch wird das Gelände mit der oben erklärten Methode komplett sichtbar.

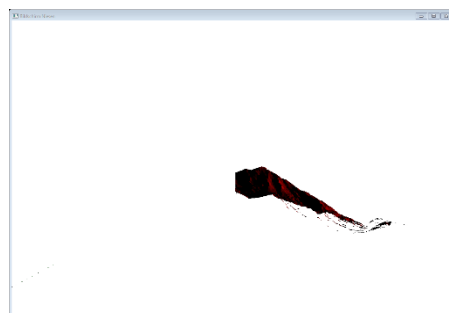


Abbildung 16 TerrainMesh mit unsichtbaren Flächen

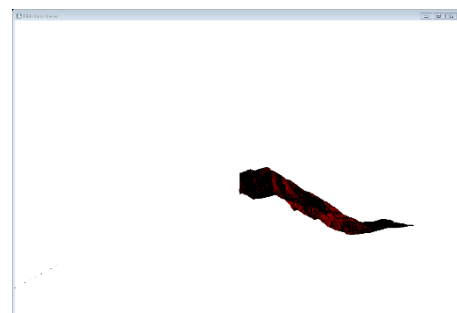


Abbildung 17 TerrainMesh mit sichtbaren Flächen



### 3.1 ARBEITEN MIT TEXCOORDS

Ein weiteres Ziel war, das Gelände einzufärben, um es anschaulicher zu gestalten.

Dafür werden die oben übersprungenen TexCoords verwendet. Mit den Tex-

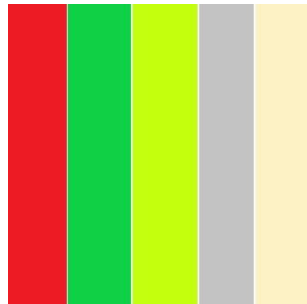


Abbildung 19 Bildvorlage für die TexCoords

Coords kann den Mesh eine Textur gegeben werden, zum Beispiel von einem Bild. Dabei werden Flächen von einem oder mehreren Bildern ausgewählt und dann den Mesh übergeben. Das Koordinatensystem des Bildes geht von 0 bis 1, das

```
//TexCoords der Dreiecke
Mesh11.getTexCoords().addAll(
    0.4f, 0.067f, //0 Rot
    0.4f, 0.133f, //1 Rot
    0.6f, 0.100f, //2 Rot
    0.4f, 0.267f, //3 Grün
    0.4f, 0.333f, //4 Grün
    0.6f, 0.300f, //5 Grün
    0.4f, 0.467f, //6 Olive
    0.4f, 0.533f, //7 Olive
    0.6f, 0.500f, //8 Olive
    0.4f, 0.667f, //9 Grau
    0.4f, 0.733f, //10 Grau
    0.6f, 0.700f, //11 Grau
    0.4f, 0.867f, //12 Schneeweiss
    0.4f, 0.933f, //13 Schneeweiss
    0.6f, 0.900f, //14 Schneeweiss
);
```

Abbildung 18 Die TexCoords im Programmcode

heisst der obere linke Punkt hat die Koordinaten (0/0) und der untere rechte Punkt die Koordina-

ten (1/1). Die Abbildung 18 zeigt, wie die TexCoords aussehen. Ich habe für jeden Streifen ein Dreieck erstellt, welche den Meshflächen zugeordnet werden. Die fünf if-then-else Anweisungen bestimmen anhand der Höhe, welche Farbe zugeteilt wird (Abbildung 19).

```
//Alle Punkte eines Quadrats
Mesh11.getPoints().addAll(
    (x11), (-yPointsF1[x11]*2), (z11), //Oberer linker Punkt P1
    ((x11) + 100), (-yPointsF2[x11]*2), (z11), //Oberer rechter Punkt P2
    (x11), (-yPointsF3[x11]*2), ((z11) + 100), //Unterer linker Punkt P3
    ((x11) + 100), (-yPointsF4[x11]*2), ((z11) + 100), //Unterer rechter Punkt P4
);

if (((-yPointsF1[x11]*2) >= -1000) && ((-yPointsF1[x11]*2) <= 0)){
    Tx1 = w + 0;
    Tx2 = w + 1;
    Tx3 = w + 2;
}
else if (((-yPointsF1[x11]*2) >= -1500) && ((-yPointsF1[x11]*2) <= -1000)){
    Tx1 = w + 3;
    Tx2 = w + 4;
    Tx3 = w + 5;
}
else if (((-yPointsF1[x11]*2) >= -2000) && ((-yPointsF1[x11]*2) <= -1500)){
    Tx1 = w + 6;
    Tx2 = w + 7;
    Tx3 = w + 8;
}
else if (((-yPointsF1[x11]*2) >= -3200) && ((-yPointsF1[x11]*2) <= -2000)){
    Tx1 = w + 9;
    Tx2 = w + 10;
    Tx3 = w + 11;
}
else{
    Tx1 = w + 13;
    Tx2 = w + 13;
    Tx3 = w + 14;
}
```

Abbildung 20 Programmcode mit den if/else Voraussetzungen

Für jede if-then-else Anweisung wird eine Farbe übergeben, welche bei einem bestimmten Fall auftreten wird. Danach werden die TexCoords sowie die Points der Methode getFaces zugeordnet und das Mesh wird gezeichnet. A1-3 sind die Punkte

```
//Fläche der beiden Dreiecke in einem Quadrat
Mesh11.getFaces().addAll(
    a1,Tx1, a2,Tx2, a3,Tx3,
    b1,Tx1, b2,Tx2, b3,Tx3,
    a3,Tx3, a2,Tx2, a1,Tx1,
    b3,Tx3, b2,Tx2, b1,Tx1
);
```

Abbildung 21 Programmcode mit den Points und TexCoords

für das erste Mesh in einem Quadrat, b1-3 die Punkte für das zweite Mesh und Tx1-3 die TexCoords, welche bei den if-then-else Anweisungen die Farbe zugeteilt erhalten.

Wie in Abbildung 15 ersichtlich ist, sind die Mesh in einer Gruppe namens root angesammelt. Wenn die Berechnungen abgeschlossen sind, wird die Gruppe, mit allen Mesh enthalten, der Scene übergeben und auf dem Bildschirm gezeichnet. Doch genau daraus entstand ein Problem, denn die TexCoords konnten der Gruppe nicht übergeben werden. Java hat dann dem Meshgelände die Defaultfarbe übergeben. Die Defaultfarbe kommt zum Einsatz, wenn keine spezifische Farbe den Mesh zugeteilt wurden. Damit das Gelände trotzdem eine Farbe hat, habe ich dem Phongmaterial die Farbe Grün zugeteilt. In Abbildung 22 ist eine Vorgängerversion des endgültigen TerrainMeshs zu sehen.

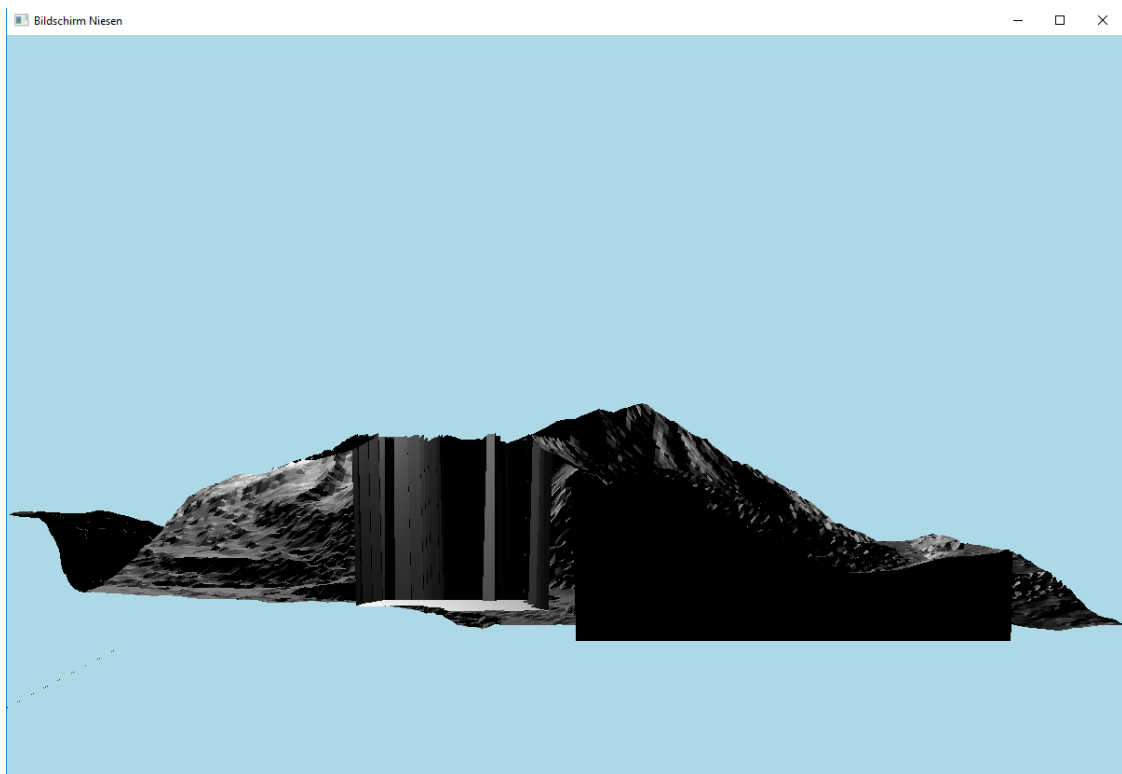


Abbildung 22 Das TerrainMesh (noch nicht als Endprodukt)

Das Gelände in Abbildung 22 ist in der Defaultfarbe dargestellt und wird erst später mit der Farbe Grün überfärbt. Im Endprodukt (Abschnitt 5.1) liegt der Gipfel des Niesens an der Stelle, an welcher man ein Renderproblem in Abbildung 22 sieht. Das Geländemodell gespiegelt darzustellen und somit den Gipfel des Niesens nicht darstellen zu können, ist darauf zurückzuführen, dass im Endprodukt das Gelände anschaulicher präsentiert werden kann. Auf das Renderproblem wird in Abschnitt 5.1 eingegangen.

## 4 DER RAY CASTER

---

Als finales Ziel meiner Arbeit wollte ich einen simplen Ray Caster von Grund auf programmieren. Im Internet habe ich bereits viele Ray Caster gefunden [5], doch weil der Ray Caster ursprünglich für Shooter-Spiele in Häusern (mit senkrechten Wänden) verwendet wurde, bringen mir diese Programmvorlagen kaum etwas. Im Buch «More Tricks of the Game», welches ich von Herr Schuppli erhalten habe, ist ein Codebeispiel eines Ray Casters in C++, doch hinsichtlich der in C++ verwendeten Methoden, aus denen ich nicht schlau geworden bin, habe ich meinen Ray Caster aus dem allgemeinen Verständnis eines Ray Casters und meinen mathematischen Überlegungen begonnen.

Basierend auf den vorhergehenden Programmen habe ich auch den Ray Caster mit verschiedenen Klassen aufgebaut. In der Klasse **Hauptprogramm** wird ein JFrame,

```
public class Hauptprogramm extends JComponent
{
    private static final long serialVersionUID = 1L;

    public static void main (String[] args)
    {
        JFrame fenster = new JFrame ("Bildschirm");
        fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fenster.setSize(600,600);
        fenster.setBackground(Color.blue);
        fenster.setVisible(true);

        RayCaster RCr = new RayCaster();
        try{
            RCr.init();
            RCr.Casten(fenster.getGraphics());
        }
        catch (Exception e){
            System.out.print(e.toString());
        }

        fenster.add(RCr);
        fenster.paintAll(fenster.getGraphics());
    }
}
```

Abbildung 23 Code der Klasse Hauptprogramm

beziehungsweise ein Bildschirmfenster erstellt. In Java ist es hilfreich, möglichst genaue Anweisungen dem Programm zu übergeben, um präzise Resultate zu erhalten. Deshalb habe ich dem Bildschirmfenster Name, Grösse und Farbe übergeben. Die Anweisung setVisible stellt sicher, dass das Bildschirmfenster angezeigt wird. Zudem wird die Klasse *RayCaster* aufgerufen,

in welcher die Berechnungen vorgenommen werden. Die try and catch Methode verlangt Java, da in Java immer Ausnahmesituationen entstehen können, die das Programm nicht verarbeiten kann. In diesem Fall hier, werden die Methoden *Init()* und *Casten()* der Methode **RayCaster** übergeben. Falls eine Ausnahmesituation eintreten sollte, würde in der Konsole eine Fehlermeldung ausgegeben werden. Die letzten beiden Zeilen übergeben dem Bildschirmfenster den **RayCaster**, sowie den Befehl, die im **RayCaster** berechneten Daten zu zeichnen.

Für den Ray Caster sind die Höhendaten in einer Map erforderlich. Deshalb habe ich die Klasse **MyMap** erstellt. In ihr werden die in Abschnitt 2.1 erwähnten Daten zusammengefügt und als eine grosse Map ausgegeben. Am einfachsten ist dies zu erreichen, in dem zuerst einen zweidimensionalen Array erstellt wird und dieser mit dem Wert 0 abgefüllt wird. Der Array hat die Masse  $2 \times \text{nColumns}$  und  $2 \times \text{nRows}$ , dies aufgrund der Grösse der vier Datenstücke mit der jeweiligen Grösse von  $\text{nColumns} \times \text{nRows}$ . In Abbildung 24 sind die Reihen und Spalten des Arrays *mapHeights[][]* mit  $2 \times \text{nColumns} + 1$  und  $2 \times \text{nRows} + 1$  angegeben. Dies aus dem Grund, dass Java bei null beginnt zu zählen und im Beispiel der Spalten der erste Quadrant von null bis  $\text{nColumns}$  reicht. Der zweite Quadrant wird an den Stellen von  $\text{nColumns} + 1$  bis  $2 \times \text{nColumns} + 1$  im Array eingefügt.

Der nächste Schritt ist, die vier Datenstücke in diesen Array einzubauen. Abbildung 24 zeigt das Kernstück der **MyMap** Klasse. In der Methode *Heights ()* wird durch

```
public void Heights() throws Exception
{
    for(a = 0; a < nColumns*2+1; a++)
    {
        for(b = 0; b < nRows*2+1; b++)
        {
            if ((a >= 0 && a < nColumns) && (b >= 0 && b < nRows)){
                mapHeights[a][b] = arrHeights1[a][b];
            }

            if ((a >= nColumns && a < nColumns*2) && (b >= 0 && b < nRows)){
                mapHeights[a][b] = arrHeights2[a-nColumns][b];
            }

            if ((a >= 0 && a < nColumns) && (b >= nRows && b < nRows*2)){
                mapHeights[a][b] = arrHeights3[a][b-nRows];
            }

            if ((a >= nColumns && a < nColumns*2) && (b >= nRows+1 && b < nRows*2)){
                mapHeights[a][b] = arrHeights4[a-nColumns][b-nRows];
            }
        }
    }
}
```

jede Stelle des Arrays *mapHeights[][]* iteriert und das korrekte Datenstück, dem jeweiligen Quadranten des Array zugeordnet.

Abbildung 24 Codeausschnitt aus der Klasse MyMap

## 4.1 RAYCASTER, DIE MAIN KLASSE

In dieser Klasse sind sehr viele Variablen enthalten. Hier sind die wichtigsten aufgelistet:

- *posx, posz*; X-, und Z-Position des Vektors
- *xs, zs*; Koordinaten des Sichtpunktes
- *xp, zp*; Koordinaten einer Pixelspalte des Bildschirms
- *DistanceToCamera*; Vordefinierte Distanz zwischen dem Bildschirm und dem Sichtpunkt (519.6 Pixel, ermöglicht 60° Blickfeld)
- *Playerwalldist*; Distanz zwischen dem Auftreffen des Strahls, auf der Map und dem Sichtpunkt
- *LengthRayScreen*; Länge des Strahls zwischen Bildschirm und Sichtpunkt
- *deltaX, deltaZ*; Differenz der jeweiligen Achsenwerte des Vektors
- *FakX, FakZ*; *deltaX, -Z* jedoch ist *FakX* = 1 und *FakZ* im Verhältnis zu *FakX*
- *lineHeight*; Die berechnete Höhe aus der Distanz und der Höhe in der Map
- *drawStart, drawEnd*; Start und Ende der Linie auf dem Bildschirm (in Y-Richtung).
- *arrHeights(1-4)[][]*; 2-dimensionaler Array mit jeweils einem Quadranten.
- *mapHeights[][]*; Alle vier *arrHeights[][]* zusammengefügt zu einem Array.
- *newMap[][]*; *mapHeights[][]* mit einem Rand von 1250 Breite (Rand ist abgefüllt mit null)

```
public void init() throws Exception
{
    Mmp.Init();
    Mmp.Heights();

    newMap = new int[xSize][ySize];

    //Neue Map erstellen (2000x2000, mit Werten 0)
    for(x = 0; x < xSize; x++)
    {
        for (z = 0; z < ySize; z++)
        {
            newMap[x][z] = 0;
        }
    }

    //Neu erstellte NewMap mit den Werten von MyMap abfüllen (MyMap in Mitte von NewMap platzieren)
    for (a = 0; a < xSize; a++)
    {
        for (b = 0; b < ySize; b++)
        {
            if ((a >= 1250 && a < 18750) && (b >= 4000 && b < 16000)){
                newMap[a][b] = Mmp.MapHeights()[a-1250][b-4000];
            }
        }
    }
}
```

Abbildung 25 Codeausschnitt der Klasse *TerrainMeshNiesen*

In der Methode *Init()* werden von der Klasse **MyMap** die beiden Methoden *Init()* und *Heights()* ausgeführt, um den Array *mapHeights[][]* zu übernehmen. In dieser

Methode wird unterhalb einen neuen endgültigen Array, *newMap[][]* erstellt. Dieser wird zuerst mit null abgefüllt, danach wird der Array *mapHeights[][]* in die Mitte des neuen Arrays *newMap[][]* platziert.

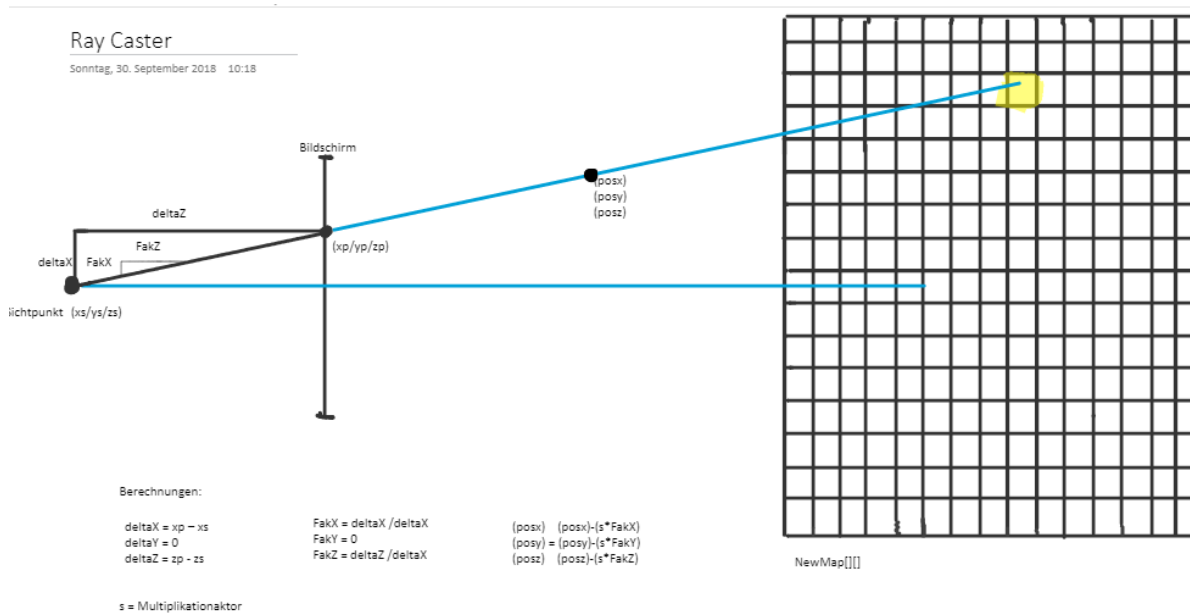


Abbildung 26 Schematische Darstellung des RayCasters

Das Grundprinzip meines Ray Casters ist in Abbildung 25 dargestellt. Der Strahl mit den Koordinaten *posx*, und *posz* geht vom Sichtpunkt aus (*xs*, *zs*) zu der aktuellen Pixelspalte. Dies basiert auf einem zwei-dimensionalen System, da die Y-Werte auf null gesetzt sind und deshalb nicht verwendet werden. Dies führt auch dazu, dass die Variablen *deltaY* und *FakY*, auch null ergeben. *deltaX* wird aus der Differenz zwischen *xp* und *xs* berechnet, aus dem gleichen Prinzip wird *deltaZ* berechnet. Um den Vektor für jeden Durchlauf möglichst klein zu vergrössern, wird *FakX* und *FakZ* verwendet. *FakX* wird den Wert 1 beziehungsweise -1 übergeben, um den kleinstmöglichen noch sinnvollen Veränderungsschritt des Vektors zu machen, sich immer um ein Feld zu verschieben. *FakZ* wird durch *FakX* dividiert, somit bleiben beide kongruent zueinander und können mit einem Faktor multipliziert werden. Mit einer while-Schleife wird die Variable *s* pro Durchlauf um den Wert eins heruntersgesetzt.

Dies hat zur Folge, dass der Strahl von hinten in der Map startet und sich pro Durchlauf der Schleife näher zum Sichtpunkt bewegt. Mit dieser Vorgehensweise werden zuerst die hinteren Geländestrukturen gezeichnet. Falls sich davor ein höherer Berg befindet, so kann die bereits gezeichnete Linie einfach überschrieben werden. Würde man den Strahl von vorne nach hinten schicken, würden selbst die kleineren Geländestrukturen gezeichnet werden, welche sich hinter einem höheren Berg befinden. Dies würde den Blick auf das Gelände verfälschen. Anschliessend werden vier if-then-else Anweisungen durchgeführt.

Der Array *newMap[][]* ist so angesetzt, dass die Positionen, an welchen die Werte zu finden sind, nicht null oder negativ werden. Dafür sind die ersten beiden if-then-else Anweisungen da: falls *posx* oder *posz* null oder kleiner sein sollte, wird *s* um den Wert eins heruntergesetzt und die Schleife beginnt von Neuem. Die dritte if-then-else Anweisung kontrolliert, ob der Vektor sich am Rand der *newMap[][]* befindet. Falls dies zutreffen sollte, wird die Variable *s* um den Wert eins heruntergesetzt und die Schleife beginnt von Neuem. Die vierte if-then-else Anweisung ist die wichtigste unter den vieren, da in ihr die Berechnungen für die Ausgabe auf dem Bildschirmfenster durchgeführt werden. Falls der Wert *newMap[][]* an der Stelle, an der sich der Vektor (*posx* und *posz*) befindet, grösser als null ist, so werden verschiedene Variablen berechnet. Die Variable *playerwalldist* wird mit der Formel von Pythagoras ausgerechnet, da die beiden Katheten, als delta x und delta z von *posx*-*xs* und *posz*-*zs* beschrieben werden können. Die Länge der Hypotenuse entspricht der Länge des Vektors. Wichtig dabei zu beachten ist, dass delta x und delta z nicht die oben erwähnten Variablen *deltaX* und *deltaZ* sind. Die hier verwendeten Variablen beinhalten die Länge delta x und delta z Werte zwischen dem Sichtpunkt und dem Auftreffpunkt des Strahls in der Map. Die if-then-else Anweisung sorgt dafür, dass diese Distanz immer positiv ist.

Der nächste Wert, welcher berechnet wird, ist die Variable *lineHeight*. Diese Rechnung kommt aus dem Kongruenzverhalten zustande, welches man in Abbildung 26 betrachten kann. Das Kongruenzverhalten von zwei Dreiecken sagt aus, dass bei verschiedener Grösse die Seiten der Dreiecke im Verhältnis zueinander unverändert bleiben. Aus diesem Verhalten berechnet sich diese Variable mit der Rechnung:

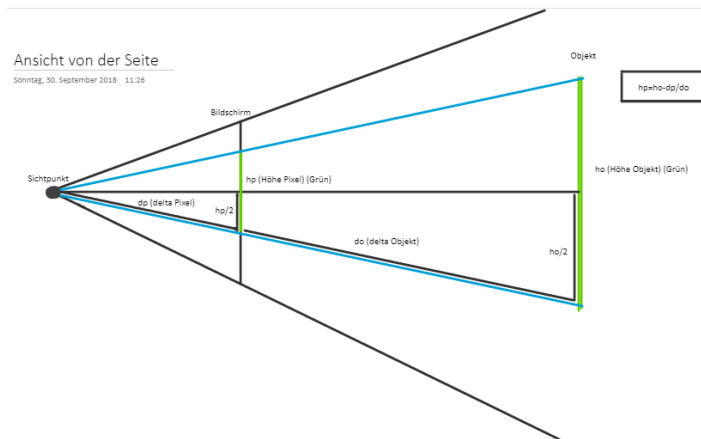


Abbildung 27 Schematische Seitenansicht des RayCasters

$$dp / (hp/2) = do / (ho/2) \rightarrow hp = dp * ho / do$$

Hat man die Variable *lineHeight* berechnet, fehlen noch die beiden Variablen *drawStart* und *drawEnd*. Diese beiden Werte bestimmen die Y-Werte der Linie auf dem Bildschirmfenster. Dabei wird vom unteren Bildschirmrand ausgegangen, und um den Wert *lineHeight* subtrahiert, welche den *drawStart* ergibt. Die Variable *drawEnd* wird um ein Pixel über dem unteren Bildschirmrand platziert. Beim *drawStart* wird eine If-Anweisungen gegeben, damit die Linie nicht ausserhalb des Bildschirmes weitergezeichnet wird.



## 5 FAZIT ZUR ARBEIT

---

Das Fazit für meine Maturarbeit fällt durchgezogen aus. Einerseits hat mir das Programmieren mit Graphics sehr viel Freude bereitet. Die kleinen Zwischenergebnisse zu sehen, war mir immer eine Genugtuung. Andererseits sehe ich im Nachhinein grössere Schwierigkeiten, eine Programmiersprache in so kurzer Zeit von Grund auf neu zu lernen. In einem Zeitplan von circa einem halben Jahr kann mit unerwarteten Zwischenfällen ein ungemein grosser Zeitdruck aufkommen. Die Vorarbeiten haben sehr viel gebracht, dank diesen Vorstufen fiel es mir leichter, die grafischen Aspekte in den Programmen auszuführen. Trotzdem haben diese Vorstufen zu viel Zeit beansprucht, da sie im Vergleich zum Terrainmesh oder zum Ray Caster kaum Code benötigten und innert wenigen Tagen geschrieben sein sollten.

### 5.1 FAZIT ZUM TERRAINMESH

Man erkennt zwar eine Geländestruktur, das Terrainmesh ist jedoch nicht restlos zufriedenstellend. An der Stelle, an welcher der Gipfel des Niesens lokalisiert wäre, erkennt man ein Renderproblem, welches mir nicht gelungen ist zu beheben. Der Output des Programms an der bestimmten Stelle lautete Null, worauf zwei mögliche Ursprungsstellen in Betracht gezogen werden können. Erstens könnte der Fehler in der Klasse **MyMap** liegen, weil in dieser Klasse den Array *Mapheights[][]* mit null abgefüllt wird. Im Zuordnen der Arrays *arrHeights1-4[][]* könnten Lücken entstanden sein. Ich habe jedoch diese Klasse besonders genau überprüft und bin mir sehr sicher, dass das Renderproblem nicht daraufhin zurückzuführen ist. Die zweite Ursprungsstelle des Problems könnte in der Klasse **TerrainMeshNiesen** liegen. In der Methode *MyMesh1()* wird durch die Spalten und Reihen des Array *newMap[][]* iteriert, dabei könnten einige Zeilen oder Spalten ausgelassen worden sein. Das Renderproblem ist jedoch nicht auf einer ganzen Zeile oder Spalte vertreten, sondern besteht aus einer beschränkten, jedoch variierender Anzahl Stellen auf einer

Zeile, welche betroffen ist. Zudem lässt die einheitliche Farbe das Modell weniger realistisch erscheinen. Auch dieses Problem konnte ich leider nicht beheben.

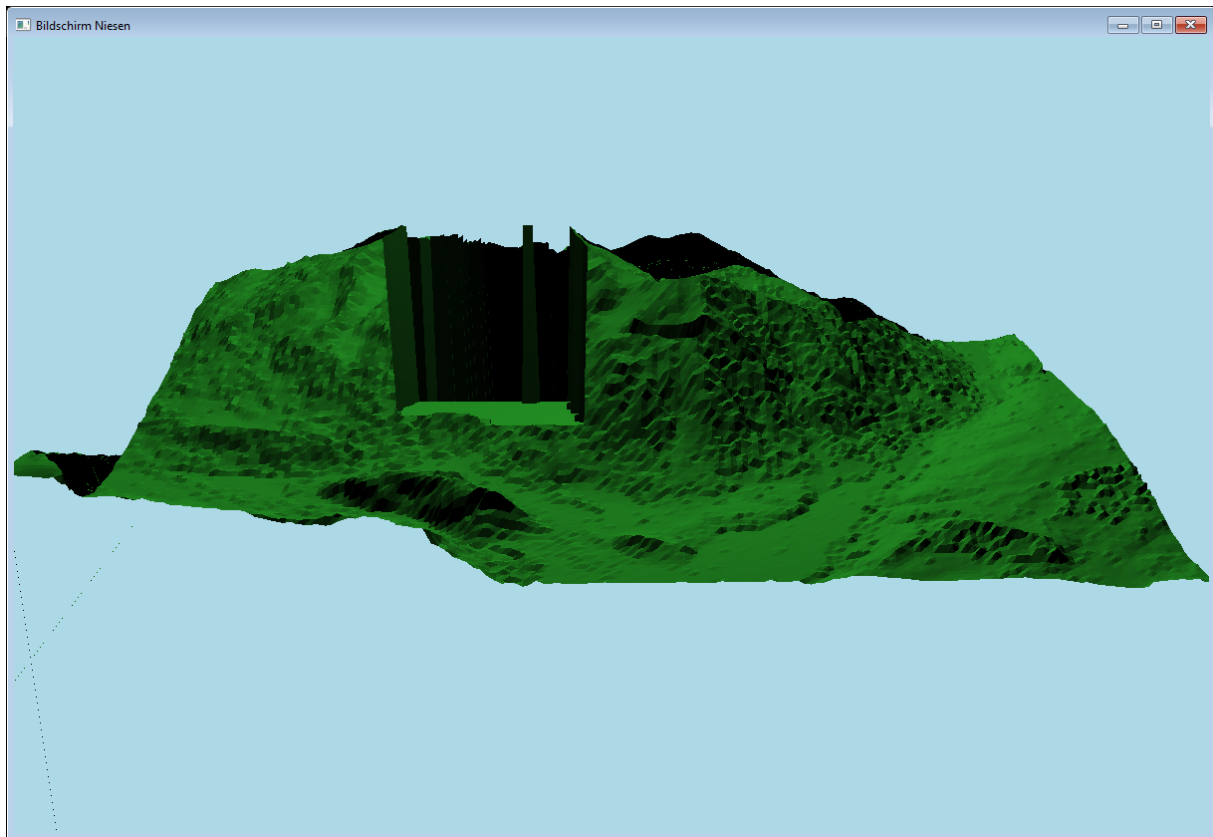


Abbildung 28 Das Endprodukt des TerrainMeshs

## 5.2 FAZIT ZUM RAY CASTER

Auch mein Ray Caster ist nicht vollkommen zufriedenstellend: Ich habe ziemlich spät mit dem Ray Caster begonnen und hatte lange Zeit Rechen- oder Logikfehler im Programm. Die Rechnungen, welche in der Klasse RayCaster ausgeführt werden, stimmen und sollten die Landschaft korrekt wiedergeben. Auf dem Bildschirm wird eine Landschaft ausgegeben, doch die senkrechten Abbrüche und die gleichmässigen Kanten sind ein Indiz, dass das Programm nicht korrekt ist. Um die Ausgabe räumlicher zu gestalten, habe ich die Farbe der Striche abhängig zur Höhe in der Map gemacht. Im Bildschirm ist jedoch, mit ein paar Ausnahmen, die ganze Landschaft Olivgrün oder Gelbgrün. Daraus lässt sich schliessen, dass die Höhe in der

Map auf einer ganzen Zeile zwischen 1200 M.ü.M. und 2000 M.ü.M. ist. Zudem sagt diese Ausgabe aus, dass dahinter keine höheren Berge zu sehen sind beziehungsweise sie hinter den Hügeln davor verborgen sind, was bei einem 2300 Meter hohen Berg wie dem Niesen ziemlich unwahrscheinlich ist.

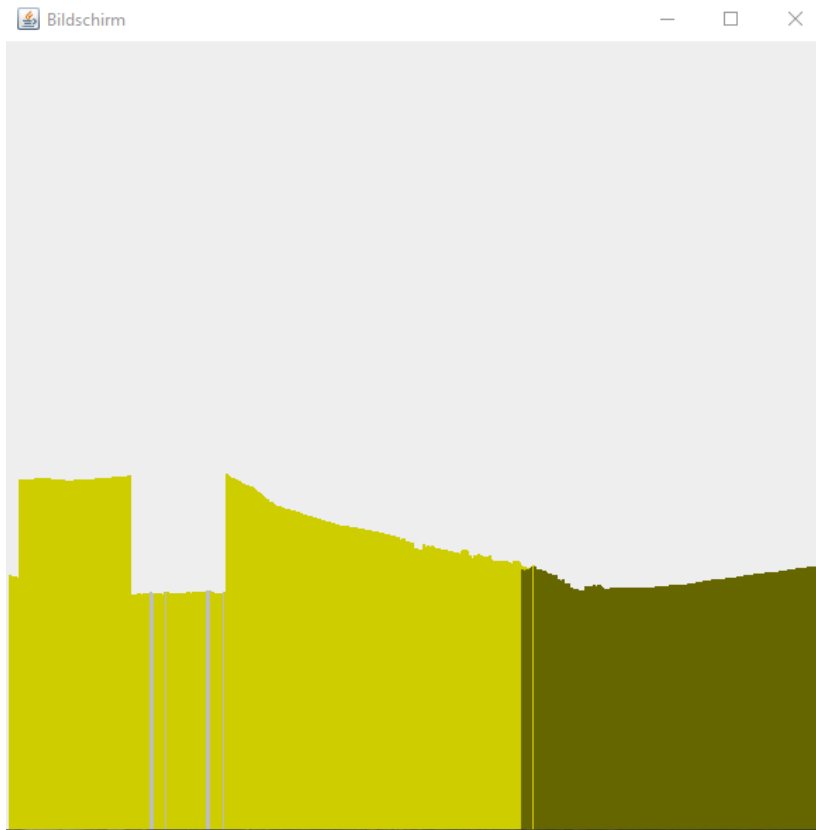


Abbildung 29 Das Endprodukt des Ray Casters

## 6 ANHANG

---

### 6.1 QUELLENVERZEICHNIS

	Deckblatt: Bild von Lotti Doswald
[1]	<a href="https://panjutorials.de/tutorials/java-tutorial-programmieren-lernen-fuer-anfaenger/">https://panjutorials.de/tutorials/java-tutorial-programmieren-lernen-fuer-anfaenger/</a> (13.10.2018)
[2]	<a href="https://www.google.ch/search?q=ray+casting&amp;rlz=1C1AVFC_enCH782CH782&amp;source=Inms&amp;tbm=isch&amp;sa=X&amp;ved=0ahU-KEwjpl7Pp99_dAhUFTlsKHZpRDn8Q_AUICigB&amp;biw=1600&amp;bih=758#img-grc=5GyE16_f-m00CM:">https://www.google.ch/search?q=ray+casting&amp;rlz=1C1AVFC_enCH782CH782&amp;source=Inms&amp;tbm=isch&amp;sa=X&amp;ved=0ahU-KEwjpl7Pp99_dAhUFTlsKHZpRDn8Q_AUICigB&amp;biw=1600&amp;bih=758#img-grc=5GyE16_f-m00CM:</a> (13.10.2018)
[3]	<a href="https://www.google.ch/search?rlz=1C1AVFC_enCH782CH782&amp;biw=1600&amp;bih=758&amp;tbm=isch&amp;sa=1&amp;ei=9vqxW_X_KoGSsAfB-ZyQBw&amp;q=ray+tracing&amp;oq=ray+tr&amp;gs_l=img..3.0.0l10.3811.4037.0.5329.2.2.0.0.0.0.127.214.1j1.2.0...0...1c.1.64.img..0.2.213...0i67k1.0.cXlm0qms2wk#img-grc=pW5cwv5pK2otrM:">https://www.google.ch/search?rlz=1C1AVFC_enCH782CH782&amp;biw=1600&amp;bih=758&amp;tbm=isch&amp;sa=1&amp;ei=9vqxW_X_KoGSsAfB-ZyQBw&amp;q=ray+tracing&amp;oq=ray+tr&amp;gs_l=img..3.0.0l10.3811.4037.0.5329.2.2.0.0.0.0.127.214.1j1.2.0...0...1c.1.64.img..0.2.213...0i67k1.0.cXlm0qms2wk#img-grc=pW5cwv5pK2otrM:</a> (13.10.2018)
[4]	<a href="http://www.geo.apps.be.ch/de/geodaten/geoprodukte-zum-download-1.html?view=sheet&amp;guid=094ce943-6ad7-4f07-aa9f-d8eb17c5cb38&amp;catalog=dlgeoproducts&amp;type=complete&amp;preview=search_list">http://www.geo.apps.be.ch/de/geodaten/geoprodukte-zum-download-1.html?view=sheet&amp;guid=094ce943-6ad7-4f07-aa9f-d8eb17c5cb38&amp;catalog=dlgeoproducts&amp;type=complete&amp;preview=search_list</a> (13.10.2018)
[5]	<a href="https://www.dummies.com/programming/java/javafx-add-a-mesh-object-to-a-3d-world/">https://www.dummies.com/programming/java/javafx-add-a-mesh-object-to-a-3d-world/</a> (13.10.2018)
[6]	<a href="https://lodev.org/cgtutor/raycasting.html">https://lodev.org/cgtutor/raycasting.html</a> <a href="https://permadi.com/1996/05/ray-casting-tutorial-table-of-contents/">https://permadi.com/1996/05/ray-casting-tutorial-table-of-contents/</a> (13.10.2018)

## 6.2 BILDVERZEICHNIS

Abbildung 1 Schematische Darstellung des Ray Casting [2] .....	6
Abbildung 2 Beispiel einer mit Ray Tracing dargestellten Szene [3] .....	6
Abbildung 3 Datensatz der «asc»-Daten im Editor.....	7
Abbildung 4 Gantrisch mit vier sichtbaren Layern .....	8
Abbildung 5 roter Würfel in JavaFX.....	9
Abbildung 6 Programmcode zum Erstellen der Box .....	9
Abbildung 7 Programmcode eines Sliders (SR3) für eine Rotation um 3600 um die Y-Achse.....	10
Abbildung 9 Veranschaulichendes Beispiel einer Pyramide mit Eckpunkten.....	11
Abbildung 8 Programmcode für eine Pyramide mit Meshs .....	11
Abbildung 10 Rote Meshpyramide zum Programmcode von oben .....	12
Abbildung 11 Programmcode der beiden Loops .....	13
Abbildung 12 Veranschaulichungs- Beispiel einer Matrix.....	14
Abbildung 13 Koordinatensystem in Java (Beispiel) .....	14
Abbildung 14 Programmcode der Methoden «getPoints» und «getFaces» .....	14
Abbildung 15 Programmcode der Zuordnung der Punkte, sowie der CullFace Funktion .....	15
Abbildung 16 TerrainMesh mit unsichtbaren Flächen.....	15
Abbildung 17 TerrainMesh mit sichtbaren Flächen.....	15
Abbildung 18 Die TexCoords im Programmcode.....	16
Abbildung 19 Bildvorlage für die TexCoords .....	16
Abbildung 20 Programmcode mit den if/else Voraussetzungen .....	16
Abbildung 21 Programmcode mit den Points und TexCoords .....	16
Abbildung 22 Das TerrainMesh (noch nicht als Endprodukt) .....	17
Abbildung 23 Code der Klasse Hauptprogramm .....	18
Abbildung 24 Codeausschnitt aus der Klasse MyMap.....	19
Abbildung 25 Codeausschnitt der Klasse TerrainMeshNiesen.....	20
Abbildung 26 Schematische Darstellung des RayCasters .....	21
Abbildung 27 Schematische Seitenansicht des RayCasters .....	23
Abbildung 28 Das Endprodukt des TerrainMeshs .....	25
Abbildung 29 Das Endprodukt des Ray Casters.....	26
Abbildung 30 Der Java Konfigurator .....	29

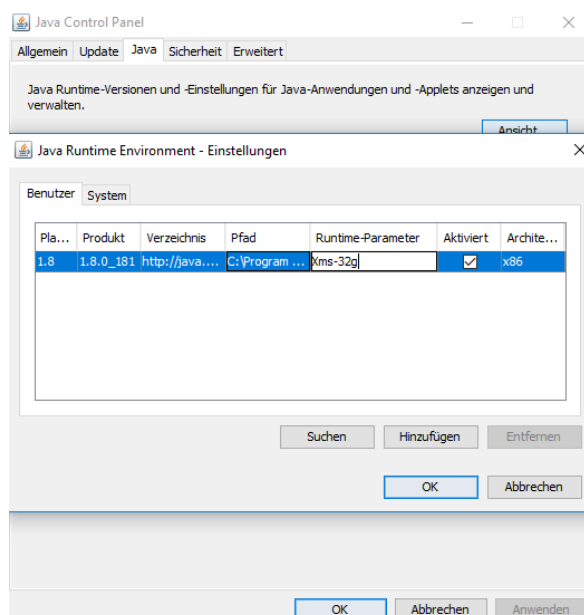
### 6.3 DIE PROGRAMME AUSFÜHREN

Um die Programme ausführen zu können müssen sie und die Datenfiles lokal auf dem Computer gespeichert werden. Die Programme wurden auf der IDE (Integrated Development Environments) Java EE, Version 1.5, geschrieben, sind jedoch kompatibel mit neueren Java RunTime Applikationen. In der Konsole muss der Befehl `java -jar "C:\Ordner\File.jar"` eingegeben werden. *Ordner* ist der Ordner, in welche die Jarfiles gespeichert wurden.

Für folgende Resultate müssen die unten aufgelisteten *Files* aufgerufen werden:

- 2-D Gantrisch: *Zeichner*
- Rote Box: *BoxZeichner*
- Pyramid Mesh: *PyramidMesh*
- Gelände Niesen: *TerrainMesh*
- Ray Caster: *Caster*

Falls die Fehlermeldung «java.lang.OutOfMemoryError: Java heap space» auftreten sollte, muss der Java Konfigurator aufgerufen werden. Im Selektionsmenu unter Java muss «Ansicht» angeklickt werden.



Unter Java muss «Ansicht» angeklickt werden. Unter «Runtime-Parameter» kann man den minimalen Arbeitsspeicher von Java erhöhen. Die Grösse kann in Megabyte und Gigabyte angegeben werden und sollte zwischen 10 und 32 Gigabyte betragen.

Abbildung 30 Der Java Konfigurator

## 7 REDLICHKEITSERKLÄRUNG

---

Ich bestätige, die vorliegende Arbeit selbständig verfasst zu haben.

Sämtliche Textstellen, die nicht von mir stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen. Die verwendeten Quellen (gilt auch für Abbildungen, Grafiken u. Ä.) sind im Literaturverzeichnis aufgeführt.

Datum:                      Unterschrift:

Silvan Spiess

## 8 ANHANG

---

### 8.1 ZEICHNER

#### 8.1.1 HauptprogrammZeichner

```
import java.io.File ;
import javax.swing.JComponent;
import javax.swing.JFrame;

public class HauptprogrammZeichner extends JComponent
{
    private static final long serialVersionUID = 1L;

    public static void main (String [] args)
    {
        //Darstellung der Daten in einem Fenster
        JFrame fenster = new JFrame ("Bildschirm");
        fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fenster.setSize(1000,1500);
        fenster.setVisible(true);

        File f = new File("DOM_1206_44.asc");
        derLeser dL = new derLeser(f);
        dL.DateiLesen();
        Zeichner zeichner1 = new Zeichner(dL.Columns(), dL.Rows(), dL.Heights());
        fenster.add(zeichner1);

        fenster.paintAll(fenster.getGraphics());
        dL.schliessen();
    }
}
```



### 8.1.2 Zeichner

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Polygon;

import javax.swing.JComponent;

public class Zeichner extends JComponent
{
    int nColumns = 0;
    int nRows = 0;
    int [][] arrHeights;

    public Zeichner (int cols, int rows, int[][] heights)
    {
        nColumns = cols;
        nRows = rows;
        arrHeights = heights;
    }

    private static final long serialVersionUID = 1L;

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        int nPoints = (nColumns / 10)+2;
        int xPoints [] = new int [nPoints];
        int zPoints [] = new int [nPoints];

        //Initialisieren der Farben bzw. additions-, subtraktionsfaktor
        int r1 = 255;
        int g1 = 255;
        int b1 = 255;
        int u1 = 20;

        //Gantrisch benennen
        Color black = new Color (0, 0, 0);
        g2.setColor(black);
        g2.drawString(("Gantrisch?! =>"), 600, 290);

        //2 Punkte für geeignetes Darstellen des Polygons
        xPoints[0] = 0;
        zPoints[0] = 800;
        xPoints[nPoints-1] = nPoints-1;
        zPoints[nPoints-1] = 800;

        //Erste Schleife, bestimmt wie viele Polygons gezeichnet werden.
        for (int b = nRows - 1; b > 0; b--){
            if ( b % 600 == 0 )
            {
                r1 = r1 - u1;
                g1 = g1 - u1;
                b1 = b1 - u1;
                Color red = new Color(r1, g1, b1);
```

```
//Zweite Schleife, zeichnet Polygon
for ( int a = 0; a < nColumns - 2; a++ ){
    if ( a % 10 == 0 )
    {
        int index = a/10+1;
        xPoints[index] = a/10;
        zPoints[index] = (2400-arrHeights[index][b]);

        Polygon poly1 = new Polygon (xPoints, zPoints, nPoints);
        g2.setColor(red);
        g2.fillPolygon(poly1);
        g2.draw(poly1);
    }
}
}
}
}
```

### 8.1.3 derLeser

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.lang.Integer;

public class derLeser
{
    Scanner s = new Scanner(System.in);

    int          nColumns = 0;
    int          nRows = 0;
    int [][]     scanHeights;

    public int Columns() { return nColumns;}
    public int Rows() { return nRows;}
    public int[][] Heights() { return scanHeights;}

    derLeser(File f){
        try{
            s = new Scanner(f);
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void DateiLesen()
    {
        String sInfoLines [] = new String [6];
        for ( int i = 0; i<6; i++)
            sInfoLines[i] = s.nextLine();

        // Zeilen 1 - 6 ausgeben
        String sColumns = sInfoLines[0].substring(6);
        String sRows = sInfoLines[1].substring(6);
        String sInfo1 = "cols=" + sColumns + ", rows=" + sRows;
        System.out.println(sInfo1);

        String xllcorner = sInfoLines[2].substring(9);
        String yllcorner = sInfoLines[3].substring(9);
        String cellsize = sInfoLines[4].substring(8);
        String nodata_value = sInfoLines[5].substring(13);
        String sInfo2 = "xllcorner=" + xllcorner + ", yllcorner=" + yllcorner + ", cellsize=" + cellsize +
            ", nodata_value=" + nodata_value;
        System.out.println(sInfo2);

        nColumns = Integer.parseInt(sColumns);
        nRows = Integer.parseInt(sRows);

        // Daten Array
        scanHeights = new int[nColumns][nRows];
```

```
// datenabfüllen in scanHeights
try {
    for ( int y = 0; y < nRows; y++ )
    {
        for ( int x = 0; x < nColumns; x++ )
        {
            if (s.hasNextFloat())
                scanHeights[x][y] = (int)s.nextFloat();
        }
    }
    System.out.println("Fertig");
}
catch (java.util.NoSuchElementException e) {
    e.printStackTrace();
}
return;
}

public void schliessen()
{
    s.close();
}
}
```

#### 8.1.4 Ausgeber

```
public class Ausgeber
{
    Ausgeber ()
    {
    }

    public String toString()
    {
        return new String();
    }
}
```

## 8.2 BoxZEICHNER

### 8.2.1 HauptprogrammBox

```
import javafx.application.Application;
import javafx.stage.Stage;

public class HauptprogrammBox extends Application
{
    public static void main (String [] args)
    {
        launch(args);
    }

    public void start (Stage Bildschirm) throws Exception
    {
        BoxZeichner bz = new BoxZeichner();
        bz.init();
        bz.BoxBewegen(Bildschirm);
    }
}
```

### 8.2.2 BoxZeichner

```
import javafx.animation.RotateTransition;
import javafx.geometry.Point3D;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
import javafx.util.Duration;

public class BoxZeichner
{
    Group root = new Group();
    PerspectiveCamera cam = new PerspectiveCamera();
    Box box = new Box(100, 100, 100);

    Point3D p1 = new Point3D(1, 0, 0);
    Point3D p2 = new Point3D(0, 1, 0);
    Point3D p3 = new Point3D(0, 0, 1);

    public void init () throws Exception
    {
        box.setMaterial(new PhongMaterial(Color.RED));
        box.setManaged(false);
    }
}
```

```
box.setTranslateX(250);
box.setTranslateY(300);
box.setTranslateZ(0);
root.getChildren().add(box);

cam.setTranslateX(-50);
cam.setTranslateY(200);
cam.setTranslateZ(500);

//Bewegung der Camera
RotateTransition Bewegung = new RotateTransition(Duration.seconds(1),cam);
Bewegung.setCycleCount(1);
Bewegung.setFromAngle(0);
Bewegung.setToAngle(-30);
Bewegung.setAutoReverse(true);
Bewegung.setAxis(p1);
Bewegung.play();
}

//Methode um Box zu bewegen
public void BoxBewegen (Stage Bildschirm) throws Exception
{
    Slider SR3 = new Slider(0, 360, 180);
    SR3.setShowTickLabels(true);
    SR3.setShowTickMarks(true);
    SR3.setMajorTickUnit(60);
    SR3.setMinorTickCount(60);
    SR3.valueProperty().bindBidirectional(box.rotateProperty());
    SR3.setTranslateX(187);
    SR3.setTranslateY(430);
    SR3.setTranslateZ(0);
    SR3.setRotationAxis(p2);
    root.getChildren().add(SR3);

    RotateTransition Bewegung = new RotateTransition(Duration.seconds(1),SR3);
    Bewegung.setCycleCount(1);
    Bewegung.setFromAngle(0);
    Bewegung.setToAngle(-30);
    Bewegung.setAutoReverse(true);
    Bewegung.setAxis(Rotate.X_AXIS);
    Bewegung.play();

    Scene scene = new Scene(root, 600, 600, Color.ANTIQUEWHITE);
    scene.setCamera(cam);

    Bildschirm.setTitle("Bildschirm");
    Bildschirm.setScene(scene);
    Bildschirm.show();
}
}
```

## 8.3 PYRAMIDMESH

### 8.3.1 HauptprogrammPM

```
import javafx.application.Application;
import javafx.stage.Stage;

public class HauptprogrammPM extends Application
{
    public static void main(String [] args)
    {
        launch(args);
    }

    public void start (Stage Bildschirm) throws Exception
    {
        PyramidMesh PM = new PyramidMesh();

        try{
            PM.init();
            PM.pyramid(Bildschirm);
        }
        catch (Exception e){
            System.out.print(e.toString());
        }
    }
}
```

### 8.3.2 PyramidMesh

```
import javafx.geometry.Point3D;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.DrawMode;
import javafx.scene.shape.MeshView;
import javafx.scene.shape.TriangleMesh;
import javafx.stage.Stage;

public class PyramidMesh
{
    public TriangleMesh pyramidMesh = new TriangleMesh();
    public Group root = new Group();
    PerspectiveCamera cam = new PerspectiveCamera();
    PointLight PL = new PointLight();
    PhongMaterial mat = new PhongMaterial();

    Point3D p1 = new Point3D(1, 0, 0);
    Point3D p2 = new Point3D(0, 1, 0);
    Point3D p3 = new Point3D(0, 0, 1);

    public void init () throws Exception
    {
        MeshView pyramid = new MeshView(pyramidMesh);

        PhongMaterial mat = new PhongMaterial(Color.RED);
        pyramid.setDrawMode(DrawMode.FILL);
        pyramid.setTranslateX(0);
        pyramid.setTranslateY(0);
        pyramid.setTranslateZ(0);
        pyramid.setMaterial(mat);
        root.getChildren().add(pyramid);

        PL.setTranslateX(200);
        PL.setTranslateY(-100);
        PL.setTranslateZ(0);
        PL.setColor(Color.WHITE);
        root.getChildren().add(PL);

        cam.setRotationAxis(p1);
        cam.setRotate(10);
        cam.setTranslateX(-300);
        cam.setTranslateY(-300);
        cam.setTranslateZ(-150);
        cam.setNearClip(0.1);
        cam.setFarClip(1000.0);
        cam.setFieldOfView(55);
    }
}
```



```
public void pyramid (Stage Bildschirm) throws Exception
{
    //TexCoords der Dreiecke
    pyramidMesh.getTexCoords().addAll(0,0);

    //Points der Dreiecke
    pyramidMesh.getPoints().addAll(
        0,    0,    0,    //P1
        0,    250,  -150, //P2
        -150,  250,  0,    //P3
        150,  250,  0,    //P4
        0,    250,  150,   //P5
    );

    //Erstellen der Dreiecke
    pyramidMesh.getFaces().addAll(
        0,0, 2,0, 1,0, //Front Links
        0,0, 1,0, 3,0, //Front Rechts
        0,0, 3,0, 4,0, //Hinten Rechts
        0,0, 4,0, 2,0, //Hinten Links
        4,0, 2,0, 1,0, //Unten Hinten
        1,0, 3,0, 4,0, //Unten Vorne
    );

    Scene scene = new Scene(root, 600, 600, Color.ANTIQUEWHITE);
    scene.setCamera(cam);

    Bildschirm.setTitle("Bildschirm");
    Bildschirm.setScene(scene);
    Bildschirm.show();
}
}
```

## 8.4 TERRAINMESH

### 8.4.1 HauptprogrammTerrainMesh

```
import javafx.application.Application;
import javafx.stage.Stage;

public class HauptprogrammTerrainMesh extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    public void start (Stage Bildschirm) throws Exception
    {
        MyMap MMp = new MyMap();
        MMp.Init();
        MMp.Heights();

        TerrainMeshNiesen TMN = new TerrainMeshNiesen(MMp.nColumns, MMp.nRows, MMp.MapHeights());
        try{
            TMN.init();
        }
        catch (Exception e){
            System.out.print(e.toString());
        }

        TMN.MyMesh1(Bildschirm);
        TMN.show(Bildschirm);
    }
}
```

### 8.4.2 derLeserCaster

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.lang.Integer;

public class derLeserCaster
{
    Scanner[] s = new Scanner[4];

    int          nColumns = 0;
    int          nRows = 0;
    int [][]     scanHeights;

    public int Columns() { return nColumns;}
    public int Rows() { return nRows;}
    public int[][] Heights() { return scanHeights;}

    derLeserCaster(File f, int index){
        try{
            s[index] = new Scanner(f);
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void DateiLesenCaster(int index)
    {
        String sInfoLines [] = new String [6];
        for ( int i = 0; i < 6; i++ )
            sInfoLines[i] = s[index].nextLine();

        // Zeilen 1 - 6 ausgeben
        String sColumns = sInfoLines[0].substring(6);
        String sRows = sInfoLines[1].substring(6);
        String sInfo1 = "cols=" + sColumns + ", rows=" + sRows;
        System.out.println(sInfo1);

        String xllcorner = sInfoLines[2].substring(9);
        String yllcorner = sInfoLines[3].substring(9);
        String cellsize = sInfoLines[4].substring(8);
        String nodata_value = sInfoLines[5].substring(13);
        String sInfo2 = "xllcorner=" + xllcorner + ", yllcorner=" + yllcorner + ", cellsize=" + cellsize +
            ", nodata_value=" + nodata_value;
        System.out.println(sInfo2);

        // Daten Array
        nColumns = Integer.parseInt(sColumns);
        nRows = Integer.parseInt(sRows);

        scanHeights = new int[nColumns][nRows];
    }
}
```

```
// datenabfüllen in scanHeights
try {
    for ( int y = 0; y < nRows; y++ )
    {
        for ( int x = 0; x < nColumns; x++ )
        {
            if (s[index].hasNextFloat())
                scanHeights[x][y] = (int)s[index].nextFloat();
        }
    }
    System.out.println("Fertig");
}
catch (java.util.NoSuchElementException e) {
    e.printStackTrace();
}
return;
}

public void schliessen (int index)
{
    s[index].close();
}
}
```

### 8.4.3 MyMap

```
import java.io.File;

public class MyMap
{
    int x;
    int y;
    int a;
    int b;

    int nCols;
    int nRows;
    int mapHeights[][];

    int arrHeights1[][];
    int arrHeights2[][];
    int arrHeights3[][];
    int arrHeights4[][];

    derLeserCaster dLC1;
    derLeserCaster dLC2;
    derLeserCaster dLC3;
    derLeserCaster dLC4;

    public int [][] MapHeights() { return mapHeights; }

    public void Init()
    {
        /*
        //Gantrisch Files
        File f1 = new File("DOM_1206_41.asc");
        File f2 = new File("DOM_1206_42.asc");
        File f3 = new File("DOM_1206_43.asc");
        File f4 = new File("DOM_1206_44.asc");
        */
        //Niesen Files
        File f1 = new File("DOM_1227_21.asc");
        File f2 = new File("DOM_1227_22.asc");
        File f3 = new File("DOM_1227_23.asc");
        File f4 = new File("DOM_1227_24.asc");

        dLC1 = new derLeserCaster(f1,0);
        dLC1.DateiLesenCaster(0);
        dLC1.schliessen(0);

        dLC2 = new derLeserCaster(f2,1);
        dLC2.DateiLesenCaster(1);
        dLC2.schliessen(1);

        dLC3 = new derLeserCaster(f3,2);
        dLC3.DateiLesenCaster(2);
        dLC3.schliessen(2);
    }
}
```

```
dLC4 = new derLeserCaster(f4,3);
dLC4.DateiLesenCaster(3);
dLC4.schliessen(3);

nColumns = dLC1.Columns();
nRows = dLC1.Rows();

mapHeights = new int[nColumns*2+1][nRows*2+1];

for (x = 0; x < nColumns*2 + 1; x++)
{
    for (y = 0; y < nRows*2 + 1; y++)
    {
        mapHeights [x][y] = 0;
    }
}

arrHeights1 = dLC1.scanHeights;
arrHeights2 = dLC2.scanHeights;
arrHeights3 = dLC3.scanHeights;
arrHeights4 = dLC4.scanHeights;
}

public void Heights() throws Exception
{
    for (a = 0; a < nColumns*2 + 1; a++)
    {
        for (b = 0; b < nRows*2 + 1; b++)
        {
            if ((a >= 0 && a < nColumns) && (b >= 0 && b < nRows)){
                mapHeights[a][b] = arrHeights1[a][b];
            }

            if ((a >= nColumns && a < nColumns*2) && (b >= 0 && b < nRows)){
                mapHeights[a][b] = arrHeights2[a-nColumns][b];
            }

            if ((a >= 0 && a < nColumns) && (b >= nRows && b < nRows*2)){
                mapHeights[a][b] = arrHeights3[a][b-nRows];
            }

            if ((a >= nColumns && a < nColumns*2) && ( b >= nRows+1 && b < nRows*2)){
                mapHeights[a][b] = arrHeights4[a-nColumns][b-nRows];
            }
        }
    }
}
```

#### 8.4.4 TerrainMeshNiesen

```
import javafx.geometry.Point3D;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.DrawMode;
import javafx.scene.shape.MeshView;
import javafx.scene.shape.TriangleMesh;
import javafx.stage.Stage;

public class TerrainMeshNiesen
{
    public Group root = new Group();
    PerspectiveCamera cam = new PerspectiveCamera();
    TriangleMesh Mesh11 = new TriangleMesh();
    MeshView MyMesh1 = new MeshView(Mesh11);
    PointLight PL = new PointLight();

    Point3D p1 = new Point3D(1, 0, 0);
    Point3D p2 = new Point3D(0, 1, 0);
    Point3D p3 = new Point3D(0, 0, 1);
    Point3D p4 = new Point3D(1, 0, 1);

    Cylinder ZyX = new Cylinder();           //X-Achse
    Cylinder ZyY = new Cylinder();           //Y-Achse
    Cylinder ZyZ = new Cylinder();           //Z-Achse

    int a1 = -4;
    int a2 = -1;
    int a3 = -3;
    int b1 = -4;
    int b2 = -2;
    int b3 = -1;

    int nColumns = 0;
    int nRows = 0;
    int [][]mapHeights;

    public TerrainMeshNiesen (int cols, int rows, int [][] heights)
    {
        nColumns = cols;
        nRows = rows;
        mapHeights = heights;
    }

    public void init () throws Exception
    {
        root.getChildren().add(MyMesh1);
```

```
//Y-Achse
ZyX.setRadius(1);
ZyX.setHeight(10000);
ZyX.setMaterial(new PhongMaterial(Color.BLACK));
ZyX.setRotationAxis(p1);
//ZyX.setRotate(90);

//X-Achse
ZyY.setRadius(1);
ZyY.setHeight(10000);
ZyY.setMaterial(new PhongMaterial(Color.BLUE));
ZyY.setRotationAxis(p3);
ZyY.setRotate(90);

//Z-Achse
ZyZ.setRadius(1);
ZyZ.setHeight(10000);
ZyZ.setMaterial(new PhongMaterial(Color.GREEN));
ZyZ.setRotationAxis(p1);
ZyZ.setRotate(90);

PL.setTranslateX(8000);
PL.setTranslateY(-15000);
PL.setTranslateZ(10000);

root.getChildren().add(ZyX);
root.getChildren().add(ZyY);
root.getChildren().add(ZyZ);
root.getChildren().add(PL);

cam.setTranslateX(8000);
cam.setTranslateY(-10500);
cam.setTranslateZ(-18000);
cam.setRotationAxis(p1);
cam.setRotate(-20);
}

public void MyMesh1 (Stage Bildschirm) throws Exception
{
    int nPoints1 = (((nColumns*2+1)-200)*((nRows*2+1)-200));
    int yPoints1P1 [] = new int [nPoints1];
    int yPoints1P2 [] = new int [nPoints1];
    int yPoints1P3 [] = new int [nPoints1];
    int yPoints1P4 [] = new int [nPoints1];

    //1. Loop für jede Zeile
    for (int z11 = 0; z11 <= (nRows*2 + 1)-200; z11++)
    {
        //2. Loop für jede Spalte (Punkt auf Zeile)
        for (int x11 = 0; x11 <= (nColumns*2 + 1)-200; x11++)
        {
            if ((x11 % 100 == 0)&&(z11 % 100 == 0))
```



```

{
    a1 += 4;
    a2 += 4;
    a3 += 4;
    b1 += 4;
    b2 += 4;
    b3 += 4;

    yPoints1P1[x11] = (mapHeights[x11][z11]);
    yPoints1P2[x11] = (mapHeights[x11 + 100][z11]);
    yPoints1P3[x11] = (mapHeights[x11][z11 + 100]);
    yPoints1P4[x11] = (mapHeights[x11 + 100][z11 + 100]);

    Mesh11.getTexCoords().addAll(0,0);

    if ((yPoints1P1[x11]) <= 0){
        yPoints1P1[x11] = (mapHeights[x11-200][z11-200]);
    }
    if ((yPoints1P2[x11]) <= 0){
        yPoints1P2[x11] = (mapHeights[x11-100][z11-200]);
    }
    if ((yPoints1P3[x11]) <= 0){
        yPoints1P3[x11] = (mapHeights[x11-200][z11-100]);
    }
    if ((yPoints1P4[x11]) <= 0){
        yPoints1P4[x11] = (mapHeights[x11-100][z11-100]);
    }
    //Alle Punkte eines Quadrats
    Mesh11.getPoints().addAll(
        (x11),                (-yPoints1P1[x11])*2,    (z11),
                                //Oberer linker Punkt P1

        ((x11) + 100),        (-yPoints1P2[x11])*2,    (z11),
                                //Oberer rechter Punkt P2

        (x11),                (-yPoints1P3[x11])*2,    ((z11) + 100),
                                //Unterer linker Punkt P3

        ((x11) + 100)         (-yPoints1P4[x11])*2,    ((z11) + 100)
                                //Unterer rechter Punkt P4
    );

    //Fläche der beiden Dreiecke in einem Quadrat
    Mesh11.getFaces().addAll(
        a1,0, a2,0, a3,0,
        b1,0, b2,0, b3,0,
        a3,0, a2,0, a1,0,
        b3,0, b2,0, b1,0
    );
}

```

```
        PhongMaterial mat = new PhongMaterial(Color.FORESTGREEN);
        MyMesh1.setDrawMode(DrawMode.FILL);
        MyMesh1.setMaterial(mat);
    }
}

public void show (Stage Bildschirm) throws Exception
{
    Scene scene = new Scene(root, 1200, 800, Color.LIGHTBLUE);
    scene.setCamera(cam);

    Bildschirm.setTitle("Bildschirm Niesen");
    Bildschirm.setScene(scene);
    Bildschirm.show();
}
```

## 8.5 CASTER

### 8.5.1 HauptprogrammCaster

```
import java.awt.Color;

import javax.swing.JComponent;
import javax.swing.JFrame;

public class HauptprogrammCaster extends JComponent
{
    private static final long serialVersionUID = 1L;

    public static void main (String[]args)
    {
        JFrame fenster = new JFrame ("Bildschirm");
        fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fenster.setSize(600,600);
        fenster.setBackground(Color.LIGHT_GRAY);
        fenster.setVisible(true);

        RayCaster RCr = new RayCaster();
        try{
            RCr.Init();
            RCr.Casten(fenster.getGraphics());
        }
        catch (Exception e){
            System.out.print(e.toString());
        }

        fenster.add(RCr);
        fenster.paintAll(fenster.getGraphics());
    }
}
```

## 8.5.2 RayCaster

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Polygon;
import java.awt.Shape;
import java.awt.geom.Line2D;

import javafx.scene.shape.Line;

import javax.swing.JComponent;

public class RayCaster extends JComponent{

    private static final long serialVersionUID = 1L;

    public double posx;           //x Pos. des Vektors
    public double posz;           //z Pos. des Vektors

    double s;                     //Faktor des Vektors
    int width = 600;              //Breite des Bildschirms
    int height = 600;            //Höhe des Bildschirms
    int u;                        //Zähler bis 600 (für die Höhe)

    int newMap[][];              //Neue grosse Map mit 0 am Rand
    int x;                       //Schlaufe
    int z;                       //Schlaufe
    int a;                       //Schlaufe
    int b;                       //Schlaufe

    int r1;                      //Rot
    int g1;                      //Grün
    int b1;                      //Blau

    MyMap MMp = new MyMap();

    public double angle;          //Winkel, zwischen Lot (auf Bildschirm) und Vektor
    public double DistanceToCamera = 519.6; //Distanz Kamera-Bildschirm (mit Blickwinkel 60°)
    public double playerwalldist; //Distanz Kamera-Hit
    public double correctplayerwalldist; //Korrigierte Distanz, zur Vermeidung vom Fish-eye Effekt
    public double LengthRayScreen; //Länge des Vektors von p-s

    public double xs = 10000.0;   //Startposition Sichtpunkt
    public double zs = 20000.0;   //Startposition Sichtpunkt

    public double xp;             //Startposition Pixel
    public double zp;             //Startposition Pixel
    public double deltaX;         //delta X Komponente des Vektors xp-xs
    public double deltaZ;         //delta Z Komponente des Vektors zp-zs
    public double FakX;           //deltaX/deltaX (kleiner Faktor um Vektor zu verlängern)
    public double FakZ;           //deltaZ/deltaZ (kleiner Faktor um Vektor zu verlängern)
```

```
double drawStart;           //Oberster Punkt der Linie auf dem Bildschirm
double drawEnd;             //Unterster Punkt der Linie auf dem Bildschirm
double lineHeight;          //Höhe des zu zeichnenden Objektes

int xSize = 20000;
int ySize = 20000;

public void Init() throws Exception
{
    MMp.Init();
    MMp.Heights();

    newMap = new int[xSize][ySize];

    //Neue Map erstellen (20000x20000, mit Werten 0)
    for (x = 0; x < xSize; x++)
    {
        for (z = 0; z < ySize; z++)
        {
            newMap[x][z] = 0;
        }
    }

    //MyMap in Mitte von NewMap platzieren
    for (a = 0; a < xSize; a++)
    {
        for (b = 0; b < ySize; b++)
        {
            if ((a >= 1250 && a < 18750) && (b >= 4000 && b < 16000)){
                newMap[a][b] = MMp.MapHeights()[a-1250][b-4000];
            }
        }
    }
}

public void Casten (Graphics g) throws Exception
{
    Graphics2D g2 = (Graphics2D)g;

    zp = zs - DistanceToCamera;

    deltaZ = DistanceToCamera;

    //iteriert durch jede Bildschirmsspalte
    for (u = 1; u < width + 2; u++)
    {
        xp = xs + ((double)u - ((double)width/2.0));

        deltaX = xp - xs;
```

```
if(deltaX <= 0){
    FakX = 1;
}
if(deltaX > 0){
    FakX = -1;
}

FakZ = deltaZ / deltaX;
if (FakZ <= 0){
    FakZ *= (-1);
}

LengthRayScreen = Math.sqrt((deltaX)*(deltaX)+(deltaZ)*(deltaZ));

posx = xs;
posz = zs;

s = xSize/3;

//Erhöht Faktor (s) des "Strahls"
while (s >= 1)
{
    posx = xs - (s * FakX);
    posz = zs - (s * FakZ);

    if (posz < 1.0){
        s--;
        continue;
    }
    if ((posx < 1.0) || (posx >= xSize+1)){
        s--;
        continue;
    }

    if (newMap[(int)posx][((int)posz)-1] <= 0){
        s--;
        continue;
    }

    if (newMap[(int)posx][((int)posz)-1] > 0){

        playerwalldist = Math.sqrt(((posx - xs)*(posx - xs))+
                                     ((posz - zs)*(posz - zs)));

        if (playerwalldist <= 0){
            playerwalldist *= (-1);
        }

        lineHeight = (newMap[(int)posx][(int)posz] *
                       LengthRayScreen / playerwalldist);
        if (lineHeight <= 0){
            lineHeight *= (-1);
        }
    }
}
```

```
drawStart = height - lineHeight;
if (drawStart < 0)
    drawStart = 0;
drawEnd = height-1;

//Grün
if ((newMap[(int)posx][((int)posz)-1] > 0)&&
    (newMap[(int)posx][((int)posz)-1] <= 1200))
    {r1 = 0; g1 = 102; b1 = 0;}

//OliveGrün
if ((newMap[(int)posx][((int)posz)-1] > 1200)&&
    (newMap[(int)posx][((int)posz)-1] <= 1500))
    {r1 = 102; g1 = 102; b1 = 0;}

//Gelbgrün
if ((newMap[(int)posx][((int)posz)-1] > 1500)&&
    (newMap[(int)posx][((int)posz)-1] <= 2000))
    {r1 = 205; g1 = 205; b1 = 0;}

//Grau
if ((newMap[(int)posx][((int)posz)-1] > 2000)&&
    (newMap[(int)posx][((int)posz)-1] <= 2500))
    {r1 = 190; g1 = 190; b1 = 190;}

//Schneeweiss
if ((newMap[(int)posx][((int)posz)-1] > 2500)&&
    (newMap[(int)posx][((int)posz)-1] <= 3500))
    {r1 = 255; g1 = 255; b1 = 204;}

Polygon poly = new Polygon();
poly.addPoint(u, (int)drawStart);
poly.addPoint(u+1, (int)drawStart);
poly.addPoint(u+1, (int)drawEnd);
poly.addPoint(u, (int)drawEnd);

Color color = new Color(r1, g1, b1);
g2.setColor(color);
g2.draw(poly);

if (u >= width + 1){
    u = 1;
}
s--;
continue;

}
else{
    s--;
    continue;
}
}
}
}
}
```