Universität Basel

# Computergrafik 2020

Prof. Dr. Thomas Vetter
Departement Mathematik und Informatik
Spiegelgasse 1, CH – 4051 Basel

Patrick Kahr (patrick.kahr@unibas.ch)
Renato Farruggio (renato.farruggio@unibas.ch)
Martin Ramm (martin.ramm@unibas.ch)

## Übungsblatt 3

**Ausgabe:** 08.04.2021
**Abgabe:** 29.04.2020

**Zu erreichende Punktzahl:** 30 (Programmieraufgaben + Theoriefragen)

*Bevor Sie die Übungen lösen, lesen Sie bitte dieses Übungsblatt sowie das Infoblatt aufmerksam durch. Es ist wichtig dass die Abgabe rechtzeitig und wie im Infoblatt beschrieben erfolgt.*

*Beachten Sie, dass zukünftige Aufgaben auf solchen, die mit einem ⋆ gekennzeichnet sind, aufbauen können. Es empfiehlt sich also, die ⋆ Aufgaben besonders sorgfältig zu lösen.*

Auf dem Übungsblatt 1 haben Sie gelernt, wie man 2D Dreiecke und Linien zeichnen und einfärben kann. Nun wollen wir einen Schritt weitergehen und 3D Objekte in ein 2D Bild zeichnen. Dazu ist es nötig, Vektoren im $\mathbb{R}^3$ korrekt auf die Bildebene zu projizieren. Ausserdem werden wir auch sehen, wie wir Transformationen wie Rotationen oder Translationen einbauen können. Wie angekündigt werden wir in diesem Aufgabenblatt auch das Konzept des Deferred Shadings implementieren, weshalb sich die Architektur der Rendering Pipeline ein wenig verändern wird.

### TestSuite

Wenn Sie `Ex3TestSuite.java` ausführen, öffnet sich eine Visuelle Test Suite, mit welcher Sie Ihre Implementierungen testen können. Nutzen Sie die TestSuite um sicher zu stellen, dass Ihre Implementierung alle Fälle abdeckt. Die Test Suite hat folgende Funktionalitäten:

- Darstellen des von Ihrer Pipeline generierten Bildes und des Gold-Standards für einen bestimmten Test (auszuwählen via das Menü [Tests]).
- Darstellen der Differenz zwischen dem eigenen und dem Gold-Standard Bild. Wenn das Differenz-Bild schwarz ist, haben Sie die Aufgabe gelöst.
- Speichern des selbst generierten Bildes.
- Je nach Test erscheint im unteren Teil ein Widget, mit welchem Sie gewisse Parameter verändern können.
- Diverse Statusmeldungen im unteren linken Bildschirmrand.

## Aufgabe 1 – Pinhole Projection ⋆ **6 Punkte**

Wir wollen einen Ausschnitt einer gegebenen 3D Welt auf den Bildschirm projizieren. Die Pinhole Projection (siehe Vorlesung) ist eine von mehreren Möglichkeiten dies zu tun. Es folgen nun einige Erklärungen zum Konzept der Pinhole Projection, bevor die eigentlichen Aufgaben vorgestellt werden.

Sei $p_w = (x_w, y_w, z_w)^T$ ein Punkt in unserer Welt, welchen wir auf die Bildebene projizieren wollen. Angenommen, das Projektionszentrum der Kamera (das Pinhole) befindet sich im Nullpunkt, die Kamera schaut in Richtung der positiven $z$-Achse und die Bildebene befindet sich im Abstand $f > 0$ parallell zur $x$-$y$-Ebene und schneidet die $z$-Achse genau im Bildmittelpunkt $(u_0, v_0)$. Dann können wir den Punkt $p_w$ wie folgt auf die Bildebene projizieren:

$$p_c =: \begin{pmatrix} x_c \\ y_c \end{pmatrix} = \frac{f}{z_w} \begin{pmatrix} x_w \\ y_w \end{pmatrix}$$

In homogenen Koordinaten schreibt sich diese Gleichung als:

$$p_c =: \begin{pmatrix} x_c \\ y_c \\ 1 \end{pmatrix} = \frac{f}{z_w} \begin{pmatrix} x_w \\ y_w \\ \frac{z_w}{f} \end{pmatrix} \sim \begin{pmatrix} x_w \\ y_w \\ \frac{z_w}{f} \end{pmatrix}$$

Die Tilde bezieht sich darauf, dass die Gleichheit nur bis auf einen Skalierungsfaktor gilt. In Matrix-Schreibweise:

$$p_c =: \begin{pmatrix} x_c \\ y_c \\ 1 \end{pmatrix} \sim \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} := \mathbf{K} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}$$

Um die Sache weiter zu vereinfachen, nehmen wir ausserdem an, dass $f = 1$ ist und bezeichnen diese Matrix mit $\mathbf{K}_0$. Die Koordinaten $(x_c, y_c)$ nennt man auch *Normalized Image Coordinates*.

Der Punkt $p_c$ befindet sich nun zwar auf der Bildebene, allerdings sind seine Koordinaten relativ zum Bildmittelpunkt $(u_0, v_0)$. Damit wir einen Punkt dem Rasterizer übergeben können, muss er aber in einem Koordinatensystem definiert sein, welches in unserem Bild oben links den Ursprung hat (wir nennen diese Koordinaten $(u, v)$). Da wir ausserdem Bilder beliebiger Grösse darstellen wollen, sollten wir auch noch eine Skalierung berücksichtigen. Den Punkt $p_i$ in Bildkoordinaten erhalten wir wie folgt:

$$p_i =: \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \begin{pmatrix} w & 0 & u_0 \\ 0 & h & v_0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_c \\ y_c \\ 1 \end{pmatrix} =: \mathbf{C} \cdot \begin{pmatrix} x_c \\ y_c \\ 1 \end{pmatrix}$$

Dabei ist $w$ die Breite und $h$ die Höhe des zu rendernden Bildes. Die Matrix $\mathbf{C}$ nennen wir auch *Kameramatrix*. Sie berücksichtigt die intrinsischen Kamera Parameter.

Die Annahme, dass sich die Kamera gerade im Weltursprung befindet, ist natürlich nicht immer gegeben (vor allem dann nicht, wenn wir ein Objekt interaktiv bewegen wollen). Damit wir die

soeben hergeleiteten Matrizen gebrauchen können, müssen wir also sicherstellen, dass sich die Kamera auch im Weltursprung befindet. Dies ist stets durch eine Kombination von Translationen und Rotationen möglich und genau die Aufgabe der *Viewmatrix*. Die Viewmatrix $\mathbf{V} \in \mathbb{R}^{4\times4}$ hat in homogenen Koordinaten folgende Form:

$$\mathbf{V} = \left(\begin{array}{c|c} \mathbf{R} & \mathbf{t} \\ \hline \mathbf{0}^T & 1 \end{array}\right)$$

Dabei ist $\mathbf{R}$ eine $3\times3$ Rotationsmatrix, $\mathbf{t} = (t_x, t_y, t_z)^T$ ein Translationsvektor und $\mathbf{0}^T = (0,0,0)$ der Nullvektor. Durch Applikation der Viewmatrix auf $p_w$ landen wir in den sogenannten *Eye Coordinates*:

$$p_e =: \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} = \left(\begin{array}{c|c} \mathbf{R} & \mathbf{t} \\ \hline \mathbf{0}^T & 1 \end{array}\right) \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}$$

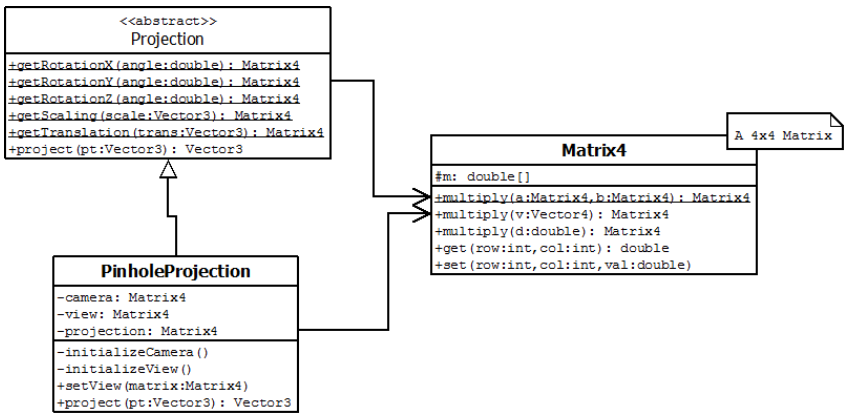
Wenn wir all dies zusammenfassen, dann sieht unsere Projektionspipeline folgendermassen aus:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \sim \mathbf{C} \cdot \mathbf{K}_0 \cdot \mathbf{V} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} = \mathbf{C} \cdot \left(\begin{array}{c|c} \mathbf{R} & \mathbf{t} \end{array}\right) \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} =: \mathbf{P} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}$$

Die $3 \times 4$ Matrix $\mathbf{P}$ nennt man auch die *Projektionsmatrix*.

Vervollständigen Sie nun die folgenden Methoden der Klasse `PinholeProjection` (eine UML-Übersicht finden Sie weiter unten):

- `initializeCamera()` welche die Kameramatrix $\mathbf{C}$ initialisiert wie soeben erklärt.
- `initializeView()` welche die Viewmatrix initialisiert. Die Implementierung dieser Methode wurde bereits vorgenommen - die Viewmatrix wird auf die Einheitsmatrix gesetzt. Überlegen Sie sich, was das für die Position der Kamera bedeutet.
- `project(Double3 pt)` welche einen dreidimensionalen Punkt projiziert. Der Rückgabewert dieser Methode ist ebenfalls vom Typ `Double3`. Die $z$-Koordinate werden wir später für den Z-Buffer gebrauchen. Für diese Koordinate müssen sie die "Inhomogenisierung", die als Folge des Einsatzes von homogenen Koordinaten nötig ist, nicht vornehmen.



Vervollständigen Sie anschliessend die folgenden statischen Methoden der Klasse `Projection`:

- `getRotationX(..)`, `getRotationY(..)` und `getRotationZ(..)` welche je eine $4 \times 4$ Rotationsmatrix um die entsprechende Achse zurückliefern.
- `getTranslation(..)` welche eine $4 \times 4$ Translationsmatrix zurückliefert.
- `getScaling(..)` welche eine $4 \times 4$ Skalierungsmatrix zurückliefert.

Benötigte Dateien: `projection.PinholeProjection.java`, `projection.Projection.java`

Mögliche Tests: Pyramid Wire Frame

Unit-Tests: `ex3.PinholeProjectionTest.java`

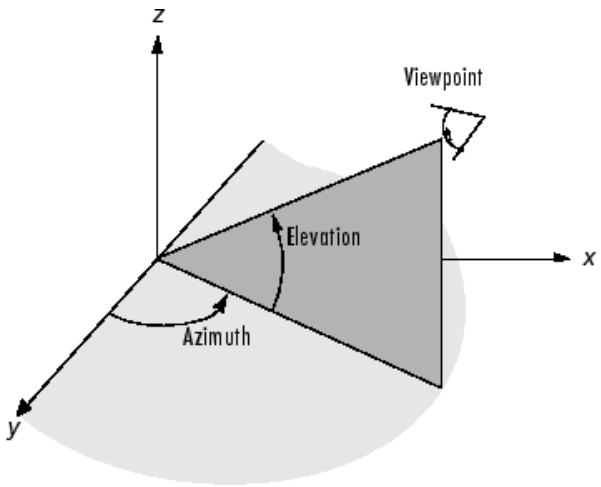
## Aufgabe 2 – Turntable

**4 Punkte**

Obwohl diese Aufgabe keinen Stern hat, empfehlen wir Ihnen, sie zu lösen. Sie ist sowohl für das Debugging von zukünftigen Aufgaben hilfreich, als auch sehr nützlich für das Verständnis der Viewmatrix.

Ziel dieser Aufgabe ist es, das gerenderte Mesh durch Ziehen der Maus drehen zu können und eine Zoom-Funktionalität einzubauen. Für das interaktive Drehen eines Objektes haben Sie in der Vorlesung das Konzept des Trackballs kennen gelernt, welches sich mit Hilfe von Quaternionen elegant implementieren lässt.

Wir verfolgen hier einen etwas abgeänderten, eher zugänglichen Ansatz. Statt eines Trackballs implementieren wir einen Turntable, der es erlaubt, ein Objekt um einen Azimuth und einen Elevations Winkel analog der folgenden Abbildung zu drehen.




Aufgepasst: die Achsenbeschriftungen in dieser Abbildund beziehen sich auf die Körperachsen des Objektes. In unserem Fall würde sich die Bildebene also parallel zur $x$-$z$-Ebene befinden.

Einen Zoom realisieren wir als eine Translation in Richtung der $z$-Achse (im Bild $y$-Achse). Die aktuelle Translation und die beiden Winkel (Azimuth und Elevation) bilden zusammen den aktuellen Zustand des Turntables. Bei Mausbewegungen werden die drei Variablen entsprechend neu belegt und die Szene neue gerendert. Vervollständigen Sie die folgenden Methoden der Klasse `Turntable` um dies umzusetzen:

- `buildViewMatrix()`: Bauen Sie hier eine Viewmatrix wie in Aufgabe 1 vorgestellt auf. Benützen Sie die statischen Methoden der Klasse `Matrix4`. Speichern Sie die fertige Viewmatrix in der Variable `currentView` ab.
- `handleAzimuth(..)` und `handleElevation(..)`: Diese Methoden werden aufgerufen wenn die Maus im GUI gedragged wird. Der übergebene Parameter `newMousePos` gibt die aktuelle $x$ bzw. $y$ Koordinate der Maus wieder. Die letzten bekannten Mauskoordinaten sind in den Members `mouseX` bzw. `mouseY` gespeichert. In diesen beiden Methoden müssen Sie aus der Mausbewegung einen Azimuth bzw. Elevations-Winkel berechnen. Berechnen Sie diese so, dass beispielsweise eine Mausbewegung vom linken bis zum rechten Rand des Bildes einer 180 Grad Drehung entspricht. Speichern Sie die neuen Winkel in den entsprechenden Membervariablen ab (ansonsten haben die schon eingefügten Funktionsaufrufe keinen Effekt).
- `zoom(..)`: Diese Methode soll das Objekt um `zoomStep` heran oder wegzoomen.

Wenn Sie diese Methoden korrekt implementiert haben, können Sie im GUI beim angegebenen Test

- durch Halten und Ziehen der Maus das Objekt drehen und
- durch Drücken von Shift und Ziehen der Maus nach oben oder unten hinein bzw. hinaus zoomen.

Bemerkung: Anstatt die Viewmatrix immer wieder neu zu berechnen, ist es auch möglich die gleiche Viewmatrix fortlaufend upzudaten, so wie Sie das in der Vorlesung gelernt haben. Dieses Verfahren führt aber meist zu numerischen Unstimmigkeiten, was wiederum seltsame Bugs zur Folge hat.

Hinweis 1: Das Objekt muss zuerst um den Azimuth und dann erst um den Elevations-Winkel gedreht werden.

Hinweis 2: Wir rechnen in Radians.

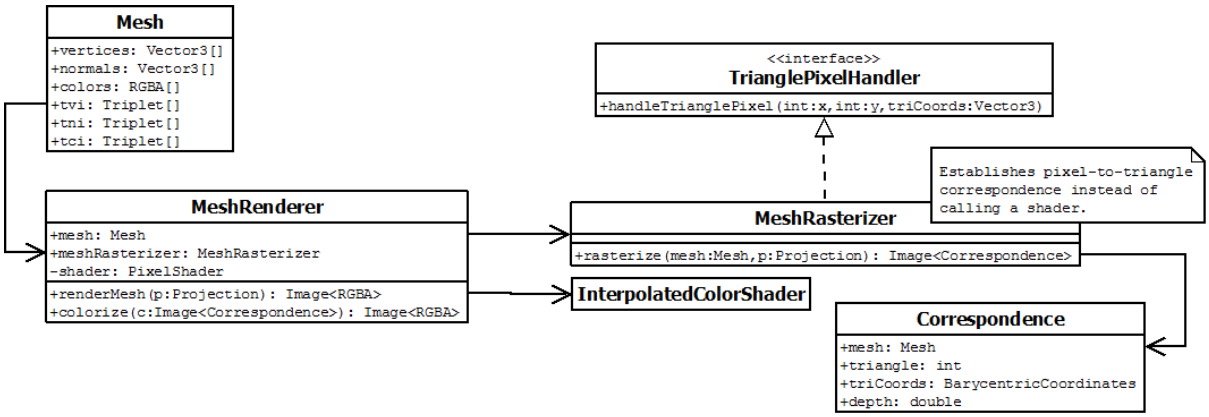
Benötigte Dateien: `projection.TurnTable.java`

Mögliche Tests: Turntable: Pyramid Wire Frame

Unit-Tests: `ex3.TurntableTest.java`

## Aufgabe 3 – Meshes, Deferred Shading und Z-Buffer **6 Punkte**

Die Pyramiden in den Tests der vorigen Aufgabe werden nur korrekt gezeichnet, weil die Dreiecke bzw. Linien in der richtigen Reihenfolge dem Renderer übergeben wurden. Diesen Missstand wollen wir beheben, indem wir einen Z-Buffer implementieren. Ausserdem implementieren wir in dieser Aufgabe einen Renderer, der dem Konzept des Deferred Shadings Folge leistet. Dazu führen wir im Wesentlichen zwei neue Datentypen ein: eine `Mesh` und eine `Correspondence` Klasse. Hier eine Übersicht der Architektur und nachfolgend die entsprechenden Erklärungen:




Ein Mesh ist nichts Weiteres als eine Ansammlung von Dreiecken. Die 3D Punkte oder Vertices dieser Dreiecke sind im Array `vertices` gespeichert. Um die Eckpunkte eines Dreiecks zu finden, benötigen wir das Array `tvi` (für `T`riangle `V`ertex `I`ndex). Dies ist ein Array aus `Triplet` und speichert die Indizes derjenigen Punkte, welche ein Dreieck bilden. `tvi` speichert nicht die 3D Punkte, sondern nur einen Index auf diese.

Um nun zum Beispiel auf den ersten Eckpunkt des 25. Dreiecks zuzugreifen, können wir folgendes tun:

```
vertices[tvi[24].get(0)];
```

`tvi[24]` gibt uns das Triplet für das 25. Dreieck und dieses Triplet gibt uns den Index, welchen wir in `vertices` nachschlagen.

Analoges gilt für die Normalen- (`tni`) und Farb- (`cni`) Informationen welche auch pro Vertex abgespeichert werden.

Der `MeshRenderer` ersetzt den bisherigen `SimpleRenderer` und funktioniert wie folgt:

Es werden jetzt keine einzelnen Linien oder Dreiecke mehr gerastert, sondern gleich ein ganzes Mesh "am Stück". Der `MeshRenderer` ruft den `MeshRasterizer` auf, der durch die Dreiecke des Meshes loopt, die Vertices projiziert und anschliessend das Dreieck wie gewohnt rastert.

Hier kommt nun das Deferred Shading ins Spiel. Der `TrianglePixelHandler`, den wir dem `TriangleRasterizer` übergeben, ist der `MeshRasterizer` selbst. Wenn der Rasterizer einen Punkt identifiziert hat, wird in der Methode `handleTrianglePixel(..)` des `MeshRasterizer`s eine Pixel-zu-Dreieck Korrespondenz gesetzt. Für jeden Bildpixel $(x, y)$ merken wir uns, welches Dreieck "hinter" diesem Pixel liegt.

Diese Korrespondenz wird durch die Klasse `Correspondence` repräsentiert. Ein `Correspondence`-Objekt besteht aus einer Referenz auf das zugehörige Mesh, die ID des Dreiecks, die baryzentrischen Koordinaten (`triCoords`) und den $z$-Wert (`depth`). Eigentlich würde die ID des Dreiecks für die Korrespondenz reichen - die restlichen Variablen erleichtern spätere Aufgaben bzw. vermeiden doppelte Berechnungen.

Um nun ein Mapping $(x, y) \mapsto \textit{Correspondence}$ für jeden Bildpunkt hinzubekommen, bedienen wir uns wiederum der Klasse `Image<T>`. Der `MeshRasterizer` gibt ein Bild vom Typ

`Image<Correspondence>` zurück. Der `MeshRenderer` nimmt anschliessend dieses Korrespondenzbild entgegen, loopt über alle eingetragenen Korrespondenzen und bestimmt mit Hilfe des bekannten `InterpolatedColorShader`s die Farbe.

Wozu der ganze Aufwand wenn wir schlussendlich wieder nur eine Farbe bestimmen? Später werden wir den `InterpolatedColorShader` austauschen, um Texturen auf ein Mesh zu rendern oder zusätzliche Berechnungen für Beleuchtung und Schatten anzustellen, bevor wir die Farbe ins fertige Bild schreiben. Durch das Deferred Shading können wir diese unterschiedlichen Aufgaben (Rasterung, Texturen, Beleuchtung, Schatten) gut auseinanderhalten und modular austauschen.

Konkret sollen Sie nun die folgenden Aufgaben lösen:

Unit-Tests: `ex3.MeshesTest.java`

### (a) Rasterization (2 Punkte)

Vervollständigen Sie die Methode `rasterize(..)` der Klasse `MeshRasterizer`. In dieser Methode müssen Sie durch alle Dreicke des gegebenen Meshes loopen, die Eckpunkte mit der gegebenen Projektion auf die Bildebene abbilden und anschliessend das projizierte Dreieck vom `TriangleRasterizer` rastern lassen. Für den TriangleRasterizer müssen zusätzliche Membervariablen zwischengespeichert werden, damit die Methode `handleTrianglePixel(..)` (siehe nächste Aufgabe) davon Gebrauch machen kann. Diese zusätzlichen Informationen sind: `currentMesh`, `currentTriangle` und die projizierten $z$-Werte des Dreiecks in `currentDepths[]`.

Benötigte Dateien: `rasterization.MeshRasterizer.java`

Mögliche Tests: Rasterization: Pyramid Mesh

### (b) Z-Buffer (2 Punkte)

Wie Sie sehen, wird die Pyramide aus (a) nicht richtig angezeigt. Dreiecke die näher an der Kamera liegen, werden von Dreiecken dahinter überschrieben.

Implementieren Sie die Methode `handleTrianglePixel(..)` der Klasse `MeshRasterizer`. Diese Methode befüllt das Korrespondenzbild. Hier sollen Sie nun auch gleich einen Z-Buffer implementieren. Das heisst, eine Korrespondenz an der Stelle $(x, y)$ wird nur dann gesetzt, wenn es entweder noch keinen Eintrag gibt oder der $z$-Wert des aktuellen Punktes kleiner ist als der eingetragene $z$-Wert. Den $z$-Wert des aktuellen anzumalenden Punktes können Sie aus den entsprechenden $z$-Werte (`currentDepths`) der Dreieckspunkte interpolieren. Unabhängig davon müssen Sie sich hier auch noch überlegen, wann ein Punkt überhaupt sichtbar sein kann, sich also nicht im "Rücken" der Kamera befindet.

Hinweis 1: Um zu schauen, wie eine Corresponendz gesetzt wird, können sie in der Klasse `MeshRasterizerWithoutZBuffer` spicken.

Hinweis 2: Um Entfernungen von Punkten zur Bildebene intuitiv verarbeiten zu können und keine Probleme mit Vorzeichen zu bekommen, multiplizieren Sie die Tiefe eines Punktes `zd` ($z$-Direction).

Benötigte Dateien: `rasterization.MeshRasterizer.java`

Mögliche Tests: Z-Buffer: Pyramid Mesh

### (c) Colorize (2 Punkte)

Nun ist das Korrespondenzbild erstellt und wir können zur Farbgebung übergehen. Vervollständigen Sie dazu die Methode `callShader(..)` der Klasse `MeshRenderer`. Diese Methode bekommt eine Korrespondenz im Bild und gibt dem Punkt über den Shader eine Farbe.

Hinweis 1: Um zu schauen, wie der Shader aufgerufen wird, können sie in der Klasse `MeshRendererWrongColorShading` spicken.

Hinweis 2: Die benötigten Farben der Eckpunkte sind im mesh gespeichert.

Benötigte Dateien: `renderer.MeshRenderer.java`

Mögliche Tests: Colorization: Pyramid Mesh

## Aufgabe 4 – Near Clipping 4 Punkte

In der Vorlesung haben Sie das Konzept des *Frustums* mit einer *near clipping* und *far clipping plane* kennen gelernt. In dieser Aufgabe sollen Sie die *near clipping plane* implementieren. Eine *far clipping plane* brauchen wir in unserer Pipeline nicht, weil wir einen `float`-wertigen Z-Buffer haben.

Wenn Sie die Methode `handleTriangePixel(..)` der vorigen Aufgabe richtig gelöst haben, dann haben Sie eigentlich schon so etwas wie ein Clipping gemacht. Nun wollen wir aber erstens die Clipping Plane auf eine beliebige Höhe setzen (repräsentiert durch die Membervariable `cNear`) und zweitens ein Dreieck gar nicht erst rastern, wenn es geclipped werden muss. Ändern Sie dazu die Methode `rasterize(..)` der Klasse `MeshRasterizer` so ab, dass ein Dreieck nur gerastert wird, wenn keiner der drei Dreiecksvertices ausserhalb der Clipping Plane liegt.

Bemerkung: Dies ist ein relativ simples Verfahren um das Clipping zu implementieren. Eleganter wäre es natürlich, nur diesen Teil des Dreiecks abzuschneiden, der auch ausserhalb der Clipping Plane liegt. Dann müsste man aber neue Dreiecke einfügen, was die ganze Sache um einiges komplizierter macht.

Benötigte Dateien: `rasterizer.MeshRasterizer.java`

Mögliche Tests: Clipping: Pyramid Mesh, Clipping: Teapot Mesh

## Aufgabe 5 – Theoriefragen — 10 Punkte

Fragen zur Theorie: Die folgenden Fragen sind schriftlich zu beantworten. Bitte legen Sie Ihre Antwort als PDF Dokument (`blatt3-antworten.pdf`) in das dafür vorgesehene Verzeichnis im Übungsframework ab.

### (a) Euler-Winkel (5 Punkte)

In der Vorlesung wurde die Orientierung eines Objektes im Raum mittels aneinander Reihen von Basisrotationen um die sogenannten Euler-Winkel eingeführt. Dabei wurden auch die Nachteile dieser Methode besprochen. Erklären Sie kurz, wie diese Darstellung mit den Euler-Winkeln funktioniert. Was passiert, wenn einer der Euler-Winkel 90° beträgt? Warum kann diese Methode bei Animationen von Bildreihen zu unerwarteten Ergebnissen führen.

Hinweis 1: Finden Sie heraus, was es mit dem sogenannten *Gimbal Lock* auf sich hat.

### (b) Quaternionen (5 Punkte)

Ein weiterer Formalismus für die Beschreibung von Rotationen sind Quaternionen. Hiermit können Rotationen um beliebige Achsen durchgeführt werden. Wie in der Vorlesung eingeführt, erhalten wir die Rotation eines Punktes $p$ um einen Winkel $\phi$ um die Achse $\vec{n}$ mit dem Operator $q()q^{-1}$, wobei $q = (\cos(\frac{\phi}{2}), \sin(\frac{\phi}{2}) \cdot \vec{n})$. Hierbei sind $p$, $q$, und $\vec{n}$ jeweils in Quaternionen Schreibweise. Den rotierten Punkt $p'$ erhalten wir also mit: $p' = qpq^{-1}$.

Machen Sie sich mit diesem Konzept vertraut. Berechnen Sie anschliessend die Rotation des Punktes $P(1, 3, 4)$ in $\mathbb{R}^3$ um 60° um die Achse $\vec{n}(\cos 30°, \sin 30°, 0)$ in $\mathbb{R}^3$.

Hinweis 1: Beachten Sie die Regeln für das Rechnen mit Quaternionen:

- $ij = -ji = k$,
- $jk = -kj = i$,
- $ki = -ik = j$

Hinweis 2: Die Inverse $q^{-1}$ einer Einheitsquaternion $q$ ist ihre Konjugierte.