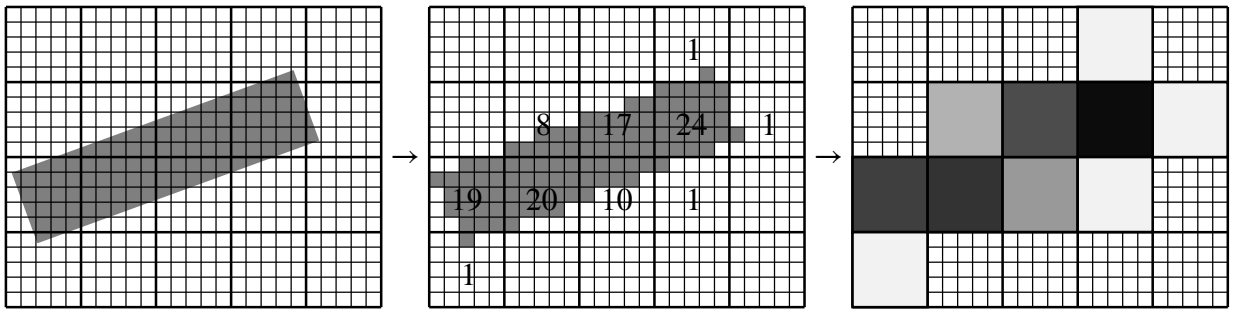
Aufgabe 1 (Ein einfaches Antialiasing-Verfahren für Linien, 8 Punkte)

Ergänzen Sie in der Datei `antial.cc` unter `/home/bildgen/Aufgaben/anti-aliasing` die Funktion
`drawAntialiasedWideLine,`

die geglättete Linien beliebiger Breite zeichnet.



Verwenden Sie hierzu ein in beide Richtungen um einen Faktor f verfeinertes Raster (in der Abbildung ist also $f = 5$). Fassen Sie die Linie in diesem feineren Raster als Rechteck auf und bestimmen Sie, welche (Sub-)Pixel eingefärbt werden müssen. Die Intensität der Pixel im Originalraster wird dann proportional zur Anzahl der eingefärbten Subpixel gewählt.

Verwenden Sie bei der Programmierung *kein* zusätzliches zweidimensionales Feld; merken Sie sich statt dessen für jede Subpixelzeile Anfangs- und Endpunkt des Rechtecks.

Testen Sie auch mit Linienbreiten < 1.

Aufgabe 2 (Strecken-Clipping nach Cohen und Sutherland, 4 Punkte)

Sei das Kappungsfenster gegeben durch das achsenparallele Rechteck $[2; 8] \times [1; 5]$ sowie Linien mit Anfangspunkt P_1 und Endpunkt P_2.

a) $P_1 = \begin{pmatrix}1\\2\end{pmatrix}, P_2 = \begin{pmatrix}10\\4\end{pmatrix}$ b) $P_1 = \begin{pmatrix}3\\3\end{pmatrix}, P_2 = \begin{pmatrix}6\\0\end{pmatrix}$

c) $P_1 = \begin{pmatrix}7\\0\end{pmatrix}, P_2 = \begin{pmatrix}10\\2\end{pmatrix}$ d) $P_1 = \begin{pmatrix}5\\6\end{pmatrix}, P_2 = \begin{pmatrix}11\\4\end{pmatrix}$

Bestimmen Sie mit dem Cohen-Sutherland-Verfahren jeweils Anfangs- und Endpunkt der gekappten Linie von Hand.

Aufgabe 3 (Strecken-Clipping nach Cyrus, Beck, Liang und Barsky, 4 Punkte)

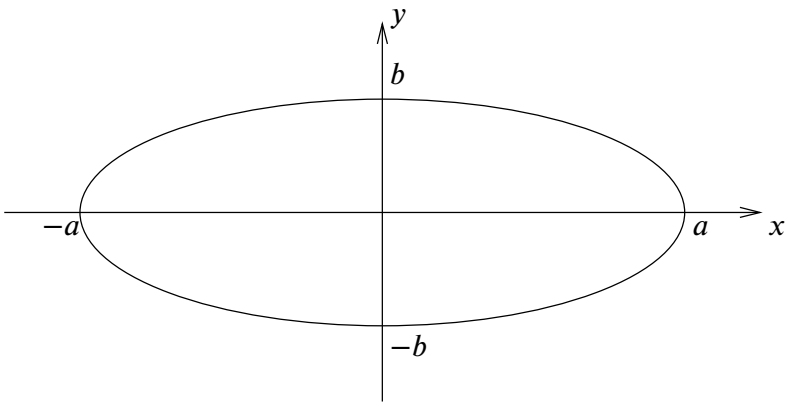
Gegeben seien das gleiche achsenparallele Rechteck und die gleichen Punkte wie in Aufgabe 2. Bestimmen Sie mit dem Cyrus-Beck-Liang-Barsky-Verfahren jeweils Anfangs- und Endpunkt der gekappten Linie. Sie können die Lösung wahlweise von Hand berechnen oder ein kleines Hilfsprogramm schreiben.

Aufgabe 4 (Gefüllte Dreiecke, 6 Punkte)

Schreiben Sie ein Programm, das drei Punkte einliest und ein ausgefülltes Dreieck zeichnet, das diese drei Punkte als Eckpunkte hat. Kommentieren Sie in Ihrem Programm, auf welcher Idee Ihr Algorithmus beruht.

Aufgabe 5 (Scan Conversion für Ellipsen, $10 = 6 + 1 + 3$ Punkte)

Die Standardellipse

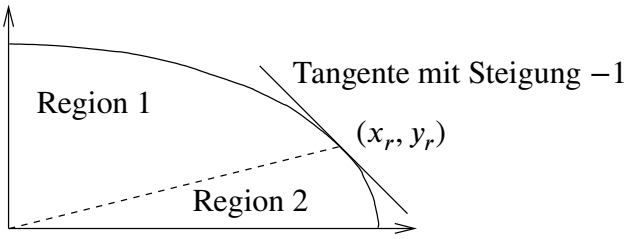


mit Mittelpunkt im Punkt $(0, 0)$ und den Halbachsen a und b ist durch die Gleichung

$$F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$$

gegeben.

Der erste Quadrant muss nun in zwei Regionen aufgeteilt werden:



Für den Punkt (x_r, y_r), an dem die Tangente Steigung -1 hat, gilt die Bedingung

$$a^2y_r = b^2x_r\,.$$

a) Formulieren Sie einen *inkrementellen Scan-Conversion-Algorithmus* für Ellipsen. Verwenden Sie zunächst für Region 1 eine Variable d_1 für die Entscheidung zwischen Ost und Südost. Am Übergang zu Region 2 muss eine neue Variable d_2 initialisiert werden für die Wahl zwischen Südost und Süd. (Ganzzahlige Rechnung ist hier nicht verlangt.)

b) Wie könnte man den Algorithmus modifizieren, damit ganzzahlig gerechnet werden kann?

c) Implementieren Sie den Algorithmus aus Teil a oder die modifizierte Variante aus Teil b, sofern Sie Teil b bearbeitet haben. Benutzen Sie eine Funktion namens `drawEllipsePoints`, die zu einem Punkt zusätzlich die anderen drei aus Symmetriegründen festgelegten Punkte zeichnet.

Ein Rahmenprogramm für die Implementierung in Teil c finden Sie im Verzeichnis `/home/bildgen/Aufgaben/ellipsen`.

Aufgabe 6 (Parallelogramm und Kreis mit Muster füllen, 8 Punkte)

Schreiben Sie Routinen

```
void parallelogramm( Drawing &pic, IPoint2D P1, IPoint2D P2,
                     IPoint2D P3, char muster,
                     const DrawColour &farbe1,
                     const DrawColour &farbe2 )
```

und

```
void kreis( Drawing &pic, IPoint2D M, int radius, char muster,
            const DrawColour &farbe1, const DrawColour &farbe2 )
```

zum Zeichnen ausgefüllter Parallelogramme und Kreise.

Dabei seien $\mathtt{P}_1 = (x1, y1)$, $\mathtt{P}_2 = (x2, y2)$, $\mathtt{P}_3 = (x3, y3)$ drei aufeinanderfolgende Ecken des Parallelogramms, wenn dieses *entgegen dem Uhrzeigersinn* durchlaufen wird, `M` sei der Mittelpunkt des Kreises und `radius` sein Radius.

Sehen Sie die Möglichkeit vor, die Objekte mit vorgegebenen Mustern zu füllen. Als Muster sollten wenigstens die folgenden symbolischen Werte zur Verfügung stehen:

- `[V]OLL`: vollständig gefüllt mit farbe1
- `[N]O`: „Nord-Ost"-schraffiert (Linien der Steigung $+1$ mit farbe1)
- `[S]O`: „Süd-Ost"-schraffiert (Linien der Steigung -1 mit farbe1)
- `[K]AROS`: „Nord-Ost"- und „Süd-Ost"-schraffiert.

Achten Sie darauf, dass das Muster *translationsinvariant* ist, d.h. beim Verschieben des Objekts wird auch das Muster „mitbewegt".

Für die nicht zum Muster gehörigen Pixel soll farbe2 verwendet werden.

Das Muster soll nicht vorberechnet werden. Stattdessen soll anhand der Position beim Zeichnen entschieden werden, ob ein Pixel im Objekt zur Schraffur (farbe1) oder zum Hintergrund (farbe2) gehört.

Hinweise:

- Ein Rahmenprogramm und Beispiel-Eingabedateien für diese Aufgabe finden Sie im Verzeichnis `/home/bildgen/Aufgaben/flood-fill`.
- Zur Bestimmung der *Ränder* der Objekte können Sie Ihre Lösungen zu den Aufgaben 5 und 6 des dritten Blattes verwenden.
- Wenn Sie Aufgabe 5 und 6 nicht bearbeitet haben, können Sie hier auch die Funktionen `Drawing::drawCircle()` und `Drawing::drawLine()` verwenden.
- Zum Füllen verwenden Sie einen modifizierten (rekursionsfreien) Flood Fill-Ansatz.
- Verwenden Sie die Funktion `Drawing::getPointColour(int x, int y)`, die eine Instanz von `DrawColour` zurückgibt, um zu testen, welche Farbe ein Pixel des Bildes hat.

Aufgabe 7 (Test des Frameworks, 0 Punkte)

Für die Bearbeitung der Übungsblätter existiert auf den Rechnern des CIP-Clusters in Raum G.14.11 ein Qt-Framework. Dieses befindet sich im Verzeichnis

```
/home/bildgen/cppqt
```

Um damit arbeiten zu können, gehen Sie wie folgt vor:

a) Falls in Ihrem Home-Verzeichnis nicht bereits ein Unterverzeichnis mit dem Namen `bin` existiert, erstellen Sie es mittels

```
cd ~
mkdir bin
```

Sie müssen sich danach neu einloggen, damit das Verzeichnis dem Suchpfad hinzugefügt wird.

b) Erzeugen Sie einen symbolischen Link auf das Compiler-Script `g++drawqt` in Ihrem soeben erzeugten Verzeichnis `~/bin`:

```
ln -s /home/bildgen/cppqt/g++drawqt ~/bin
```

c) Ein Beispielprogramm finden Sie unter

```
/home/bildgen/cppqt/bsp1.cc
```

Kopieren Sie das Beispielprogramm in Ihr Home-Verzeichnis und rufen Sie das Compiler-Script mit der Quelltext-Datei des Beispiels als Parameter auf, um es zu kompilieren:

```
cp /home/bildgen/cppqt/bsp1.cc .
g++drawqt bsp1.cc
```

Erlaubte Dateiendungen sind `c`, `cc`, `cpp` und `c++`.

d) Das erzeugte Programm trägt den Namen der Quelltextdatei ohne deren Endung. Testen Sie das Beispielprogramm:

```
./bsp1
```

Eine vollständige Auflistung der Funktionalität des Qt-Frameworks finden Sie unter:

```
/home/bildgen/cppqt/doc/index.html
```

Es existiert noch ein weiteres Compiler-Script namens `g++drawqt-g`, welches mit Debug-Symbolen compiliert.

Aufgabe 8 (Funktionenplotter, 4 Punkte)

Schreiben Sie ein Programm, das nacheinander folgende Funktionen $f_i : \mathbb{R} \to \mathbb{R}$ mit

$$\begin{aligned} f_0(x) &:= \frac{1}{2} + \frac{1}{2}x & x &\in [-5, 5] \\ f_1(x) &:= \sin(x) & x &\in [-5, 5] \\ f_2(x) &:= \sqrt{x} & x &\in [0, 4] \\ f_3(x) &:= \frac{1}{x} & x &\in [0.2, 3.0] \end{aligned}$$

auf den entsprechenden Intervallen graphisch darstellt.

In der Datei `plotter.cc`, welche im Verzeichnis `/home/bildgen/Aufgaben/plotter` erhältlich ist, finden Sie eine Vorlage, in der Sie nur die Funktion `plotter` ergänzen müssen. Verwenden Sie zum Zeichnen ausschließlich die Funktion `Drawing::drawPoint()`.

Skalieren Sie die Darstellung so, dass für das Intervall $X = [x_l, x_r]$ die linke Grenze x_l auf den linken Fensterrand und x_r auf den rechten Fensterrand fällt. Weiterhin soll vertikal so skaliert werden, dass $\min_{x \in X}(f_i(x))$ auf den unteren und $\max_{x \in X}(f_i(x))$ auf den oberen Fensterrand fallen.

Zeichnen Sie weiterhin die Achsen $\{(x, y) : y = 0\}$ bzw. $\{(x, y) : x = 0\}$, falls sie im sichtbaren Bereich liegen.

Aufgabe 9 (Objekterzeugung mit Grammatiken, Blumenwiese, 5 Punkte)

Erzeugen Sie eine Blumenwiese, auf der sich jedes Pflänzchen gemäß dem Beispiel aus Abschnitt 8.3.2 der Vorlesung entwickelt. Wählen Sie bei mehreren Ableitungsmöglichkeiten jeweils passende Wahrscheinlichkeiten, so dass sich insbesondere ein sinnvoller Anteil an Knospen weiterentwickelt. Beschränken Sie sich auf eine 2D-Darstellung und animieren Sie das Wachstum.

Aufgabe 10 (Einfache Kreise, 3 Punkte)

Schreiben Sie ein Programm, das

1. ein zunächst leeres Bild erzeugt,
2. einen Mittelpunkt *m* und einen Radius *r* einliest und
3. einen gefüllten Kreis um *m* mit Radius *r* malt.

Denken Sie sich hierzu einen möglichst einfachen Algorithmus zum Zeichnen von Kreisen aus und kommentieren Sie in Ihrem Programm, auf welcher Idee Ihr Algorithmus beruht.

Verwenden Sie nur die Funktion `Drawing::drawPoint()` und *nicht* `Drawing::drawCircle()`.

Aufgabe 11 (Scan Conversion für Kreise, 6 Punkte)

Schreiben Sie eine Funktion

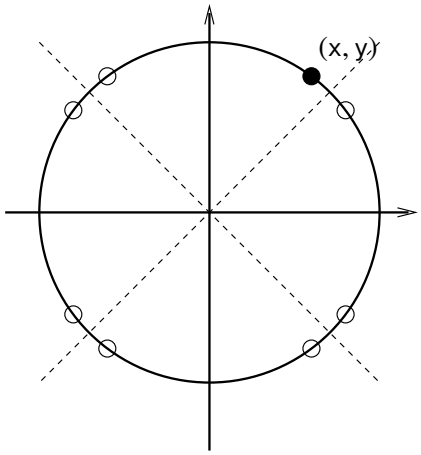
```
void drawCircle(Drawing& pic, IPoint2D center, int radius, bool filled,
                int colour = 0)
```

die einen Kreis um den Punkt `center` mit Radius `radius` zeichnet. Im Falle `filled = true` soll der Kreis ausgefüllt werden.

Implementieren Sie hierzu einen *inkrementellen, ganzzahligen Scan-Conversion-Algorithmus* zum Zeichnen der Kreislinie und verwenden Sie eine Hilfsfunktion

```
void drawCirclePoints(Drawing& pic, int x, int y, IPoint2D center,
                      bool filled, int colour = 0)
```

welche zu einem Punkt `(x, y)` im zweiten Oktanten alle hierdurch aus Symmetriegründen festgelegten Punkte zeichnet oder verbindet, vergleiche Skizze.



Ein Rahmenprogramm finden Sie unter `/home/bildgen/Aufgaben/kreise`.

Aufgabe 12 (Einfache Linien, 3 Punkte)

Schreiben Sie ein Programm, das

1. ein zunächst leeres Bild erzeugt,
2. zwei Punkte einliest,
3. eine Linie zeichnet, die beide Punkte verbindet,

4. und 2 bis 3 solange wiederholt, bis der Anwender negative Koordinaten eingibt.

Verwenden Sie hierzu nur die Funktion `Drawing::drawPoint()` und *nicht* `Drawing::drawLine()`.

Verwenden Sie einen möglichst einfachen Algorithmus und kommentieren Sie in Ihrem Programm, wie Ihr Algorithmus funktioniert.

Aufgabe 13 (Scan Conversion für Linien, 4 Punkte)

In der Datei `linien.cc` im Verzeichnis `/home/bildgen/Aufgaben/linien` auf dem CIP-Cluster finden Sie eine halb-fertige Funktion `drawLine`, die Geraden mit Steigungen $-1 \leq m \leq 1$ mittels *inkrementeller, ganzzahliger Scan Conversion* malt. Ergänzen Sie die Fälle $m < -1$ und $m > 1$. Sie können zum Testen die Eingabedatei `linien.in` verwenden. Für diese Beispiel-Eingabedatei müssen die letzten vier grauen Linien die schwarzen, in umgekehrter Richtung verlaufenden, genau überdecken.

Aufgabe 14 (Unterbrochene Linien, 6 Punkte)

Erweitern Sie das Rahmenprogramm `ulinien.cc`, welches Sie im Verzeichnis `/home/bildgen/Aufgaben/ulinien` finden können, so dass unterbrochene Linien gezeichnet werden. Dabei ist `std::vector<float> mask` die Maske, die definiert, an welchen Stellen die Linie unterbrochen wird.

Beachten Sie, dass die gegebene Maske nur für horizontale und vertikale Linien verwendet werden kann. Überlegen Sie sich, wie Sie eine steigungsunabhängige Strichlänge realisieren können und implementieren Sie Ihr Programm entsprechend. Das Beispielbild im Verzeichnis `/home/bildgen/Aufgaben/ulinien` zeigt, wie eine Implementierung mit steigungsunabhängigen Strichlängen aussehen *kann*.

Aufgabe 15 (Bézier-Kurven mit OpenGL, 4 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/opengl-6` finden Sie eine OpenGL-Implementierung der Bézier-Kurven aus Aufgabe 40. Ergänzen Sie in der Funktion

```
void maleBezier( const vector<DPoint2D>& points, int nSegments )
```

das Zeichnen der Kurvenstücke. Gehen Sie dafür wie folgt vor:

1. Legen Sie mittels `glMap1f` und des Target-Parameters `GL_MAP1_VERTEX_3`, welcher dabei anzugeben ist, die Kontrollpunkte des aktuellen Kurvenstücks fest.
2. Aktivieren Sie die Kontrollpunkte mittels `glEnable`.
3. Teilen Sie OpenGL mit, dass durch Linien verbundene Punkte gezeichnet werden sollen. Dies erfolgt mit dem Befehl `glBegin( GL_LINE_STRIP );`
4. Werten Sie die Bézier-Kurve mittels `glEvalCoord1f` an Zwischenpunkten aus, so dass pro Kurvenstück `nSegments` Linien entstehen.
5. Beenden Sie das Zeichnen mit `glEnd`.

Informationen zu den benötigten Befehlen erhalten Sie auf
`http://www.opengl.org/sdk/docs/man/`

Sie können zum Testen die Dateien `points?.in` benutzen.

Aufgabe 16 (Bézier-Flächen mit OpenGL, 2 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/opengl-4` finden Sie eine OpenGL-Implementierung der Bézier-Flächen aus Aufgabe 41. Ergänzen Sie in der Funktion

```
void zeichneBezierFlaeche( const vector<vector<Vec3D> >& p,
                           int nMeshSize = 10 )
```

das Zeichnen der Flächenstücke. Gehen Sie dafür wie folgt vor:

1. Legen Sie mittels `glMap2f` und des Target-Parameters `GL_MAP2_VERTEX_3`, welcher dabei anzugeben ist, die Kontrollpunkte des aktuellen Flächenstücks fest.
2. Aktivieren Sie die Kontrollpunkte mittels `glEnable`.
3. Erzeugen Sie unter Verwendung des Befehls `glMapGrid2f` ein Mesh, das aus `nMeshSize` Partitionen in jeder Richtung besteht.
4. Zeichnen Sie die Bézier-Fläche mit `glEvalMesh2`.

Im Gegensatz zu Aufgabe 41 wird hier nicht zwischen `anzkurv` und `anzlin` unterschieden sondern es gibt nur einen Parameter `nMeshSize` für die Feinheit des Gitters.

Informationen zu den benötigten Befehlen erhalten Sie auf
`http://www.opengl.org/sdk/docs/man/`

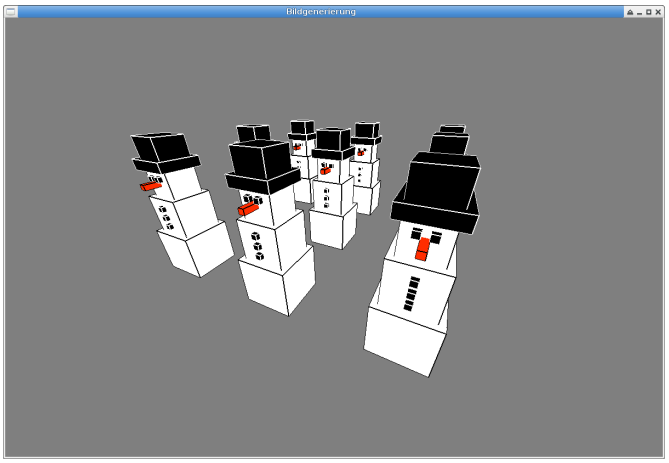
Aufgabe 17 (Modellierung mit OpenGL, 4 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/opengl-1` finden Sie ein Rahmenprogramm zu dieser Aufgabe, das die Initialisierung von `OpenGL` für Sie übernimmt. Mittels eines Aufrufs von `make` können Sie es kompilieren.

Ergänzen Sie das Zeichnen einer Gruppe von Schneemännern. Vervollständigen Sie dafür die Funktionen `drawSnowman` und `drawSnowmen`.

Die einzelnen Teile der Schneemänner sollen aus Quadern bestehen. Dafür existiert bereits eine Funktion namens `drawCube(const DrawColour& col)`, die einen Würfel mit den Eckpunkten $(-1,-1,-1)$ und $(1,1,1)$ sowie der übergebenen Farbe zeichnet. Mit `glScalef` und `glTranslatef` können Sie die Würfel skalieren und verschieben. Bei der Skalierung kann man für jede Koordinatenrichtung einen eigenen Faktor angeben, so dass man den Würfel in einen Quader transformieren kann. Weiterhin werden Sie die Befehle `glPushMatrix` und `glPopMatrix` an geeigneten Stellen benötigen.

Unten sehen Sie eine Beispielausgabe einer Lösung dieser Aufgabe.



Aufgabe 18 (Perspektivische Projektion mit OpenGL, 2 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/opengl-2` finden Sie eine OpenGL-Implementierung der perspektivischen Projektion aus Aufgabe 30. Compilieren lässt sich das Programm mittels `make`. Weiterführende Informationen zu OpenGL finden Sie auf der Webseite `http://www.opengl.org/sdk/docs/man/`.

OpenGL nutzt im Wesentlichen zwei Matrizen, um Objekte, welche in Objektkoordinaten gegeben sind, in normalisierte Koordinaten zu transformieren und anschließend zu clippen und zu zeichnen. Zum ersten handelt es sich dabei um die sogenannte ModelView-Matrix, welche die Objektkoordinaten in Augenkoordinaten überführt. Diese Matrix übernimmt also die Transformation des Augenpunktes (gegeben durch `COP`) in den Koordinatenursprung mit Blickrichtung (gegeben durch `-VPN`) entlang der negativen z-Achse, so wie es auf Seite 4-12 des Skriptes dargestellt ist. Die zweite Matrix ist die Projektionsmatrix, die die Augenkoordinaten im Unterschied zur Vorlesungen in einen Bildraum

$$B = \left\{(x, y, z)^T \mid -1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 1\right\}$$

transformiert, an dem geclippt werden kann und welcher sich einfach auf den Bildschirm (den sogenannten Viewport) projizieren lässt.

Die Projektionsmatrix ist im Rahmenprogramm bereits mittels `glFrustum` gegeben und Ihre Aufgabe besteht zunächst darin, die ModelView-Matrix korrekt zu setzen. Verwenden Sie hierfür den Befehl `gluLookAt` der OpenGL Utility Library, deren Dokumentation Sie auch auf `http://www.opengl.org/sdk/docs/man/`

finden. Sobald dies implementiert ist, sollte Ihr Programm bereits—bis auf das Clipping—die gleichen Bilder erzeugen, wie das Programm zu Aufgabe 30. Das Clipping in z-Richtung funktioniert ebenfalls bereits.

Um nun identische Bilder zur Lösung von Aufgabe 30 zu erhalten, muss ein künstliches Clipping auf die durch `clip.uminf`, `clip.umaxf`, `clip.vminf` und `clip.vmaxf` gegebene Ausdehnung auf der Projektionsebene erfolgen.

Ist die Projektionsebene gegeben durch [`umin`, `umax`] × [`vmin`, `vmax`], ergibt sich der geclippte Bereich als [`umin` · `clip.uminf`, `umax` · `clip.umaxf`] × [`vmin` · `clip.vminf`, `vmax` · `clip.vmaxf`].

Korrektes Clipping kann auf zwei Arten erzielt werden:

1. Mittels `glClipPlane` können Sie vier Clipping-Ebenen definieren, die durch den Koordinatenursprung verlaufen und den Sichtbereich in x- und y-Richtung auf der Projektionsebene auf den oben erwähnten geclippted Bereich einschränken. Hierfür müssen die Normalenvektoren der Ebenen in Abhängigkeit von `znear` und der Größen des geclippten Bereiches ermittelt werden.
2. Das Sichtfenster (Viewport) kann entsprechend beschränkt werden. Anschließend muss die Berechnung der Projektionsmatrix (`glFrustum`) angepasst werden, so dass die Projektion nun für den geclippten Bereich stattfindet.

Testen Sie Ihr Programm mit den gleichen Eingabedateien wie auch Aufgabe 30.

Aufgabe 19 (Rotationskörper mit OpenGL, 4 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/opengl-5` finden Sie eine OpenGL-Implementierung der Rotationskörper aus Aufgabe 46. Ergänzen Sie in der Funktion

```
void zeichneRotationskoerper( const vector<Vec3D>& p, RotkDaten daten )
```

das Zeichnen der Bézierkurven und Kreise, die den Rotationskörper darstellen.

Aufgabe 20 (Hidden-Surface-Verfahren und Beleuchtung mit OpenGL, 1 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/opengl-3` finden Sie eine OpenGL-Implementierung der Programme aus den Aufgaben 31 und 32. Aktivieren Sie die Verwendung des z-Buffer-Verfahrens mittels der Befehle `glEnable` und `glDepthFunc` mit passenden Parametern.

Informationen dazu finden Sie auf der Webseite
`http://www.opengl.org/sdk/docs/man/`

Aufgabe 21 (Painter's Algorithm, 10 Punkte)

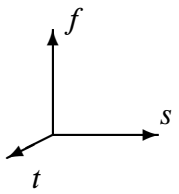
Der *Painter's Algorithm* ist ein besonders einfacher Algorithmus zur Elimination verdeckter Kanten und Flächen. Er ist anwendbar, wenn die Objekte sich nicht gegenseitig durchdringen, und basiert auf folgender Strategie: Man zeichnet einfach *alle* Objekte (Polygone müssen dabei ausgefüllt werden), wobei *„von hinten nach vorne“* gearbeitet wird. Dadurch werden die durch weiter vorne liegende Objekte verdeckten (und damit unsichtbaren) Teile der hinteren Objekte automatisch „übermalt“.

Stellen Sie mit Hilfe des Painter's Algorithm den Graphen der Funktion

$$f(s,t) = 10e^{-\left(\frac{s^2}{25}+\frac{t^2}{100}\right)}$$

auf dem Bereich $(s,t) \in [-5;5] \times [-5;5]$ dar.

Approximieren Sie dabei den Graphen durch ein geeignetes Netz von Dreiecken und verwenden Sie zur Darstellung die sogenannte *Kabinett-Projektion*.



Dabei handelt es sich um eine Parallelprojektion, wobei die s- und die f-Achse waagerecht bzw. senkrecht dargestellt werden und die t-Achse um 30° geneigt und um den Faktor ½ verkürzt ist.

Verwenden Sie das Rahmenprogramm `painters.cc` unter `/home/bildgen/Aufgaben/painters` und implementieren Sie die Funktionen:

```
Matrix4x4 berechneMpar(double umin, double umax, double vmin, double vmax,
                       double& ratio)
void erzeugeFlaeche(double xmin, double xmax, double zmin, double zmax, int num,
                    const std::function<double(double, double)>& func,
                    std::vector<Dreieck>& dreiecke )
```

Aufgabe 22 (Partikelsysteme, Feuer, 4 Punkte)

Erzeugen Sie eine Animation eines lodernden Feuers gemäß Beispiel 8.8 der Vorlesung. Eine 2D-Darstellung genügt.

Aufgabe 23 (Gefüllte Polygone, 10 Punkte)

Schreiben Sie eine Funktion

```
void drawFilledPolygon(Drawing& pic, const vector<IPoint2D>& ecken,
                       int colour = 0)
```

die mittels Algorithmus 3.22 der Vorlesung ein ausgefülltes Polygon zeichnet. Durchlaufen Sie die Zeilen y_{min} bis y_{max}, die das Polygon enthalten, und datieren Sie in jedem Schritt eine *Tabelle der aktiven Kanten* auf. Dies ist eine Liste, die zu jeder Kante, die die aktuelle y-Scan-Line schneidet, deren oberen Endpunkt, den aktuellen x-Wert sowie die Steigung enthält. Die Kanten werden nach x und ggf. nach Steigung sortiert gespeichert.

Im Verzeichnis `/home/bildgen/Aufgaben/polygone-1` finden Sie ein Rahmenprogramm und einige Eingabedateien.

Aufgabe 24 (Gefüllte Polygone, 4 Punkte)

Ändern Sie Ihre Polygon-Funktion aus Aufgabe 23 (oder die Musterlösung, welche Sie im Verzeichnis `/home/bildgen/Aufgaben/polygone-2` finden können) so ab, dass das Polygon nicht einfarbig, sondern mit einem Muster gefüllt wird. Ein Rahmenprogramm sowie Dateien mit Mustern und Beispieleingaben finden Sie im Verzeichnis `/home/bildgen/Aufgaben/polygone-2`.

Sorgen Sie dafür, dass die Füllung unabhängig davon ist, an welcher Stelle des Bildes sich das Polygon befindet. D.h. das Polygon soll sich wie ein bemaltes Stück Papier verhalten und nicht wie eine Loch-Schablone, die vor einem feststehenden bemalten Hintergrund bewegt wird. Dies können Sie mittels der Eingabedateien `polygmuster1.in` bis `polygmuster3.in` testen. In allen drei Füllungen sollte in der linken unteren Ecke des Dreiecks ein blaues Quadrat des Musters zu sehen sein.

Der Befehl `pic.drawPoint(x, y, QColor(muster.pixel(xmuster, ymuster)))` malt einen Punkt in der Farbe, die das Muster `muster` an der Stelle (`xmuster`, `ymuster`) enthält, an die Stelle (`x`, `y`) ins Bild `pic`. Mit `muster.width()` und `muster.height()` können Sie Breite und Höhe des Musters ermitteln.

Aufgabe 25 (Gefüllte Polygone, 12 Punkte)

Schreiben Sie eine Funktion

```
void drawPatternPolygon( Drawing& pic, const vector<IPoint2D>& ecken,
                         const QImage& muster )
```

die mittels Algorithmus 3.22 der Vorlesung ein mit einem Muster gefülltes Polygon zeichnet.

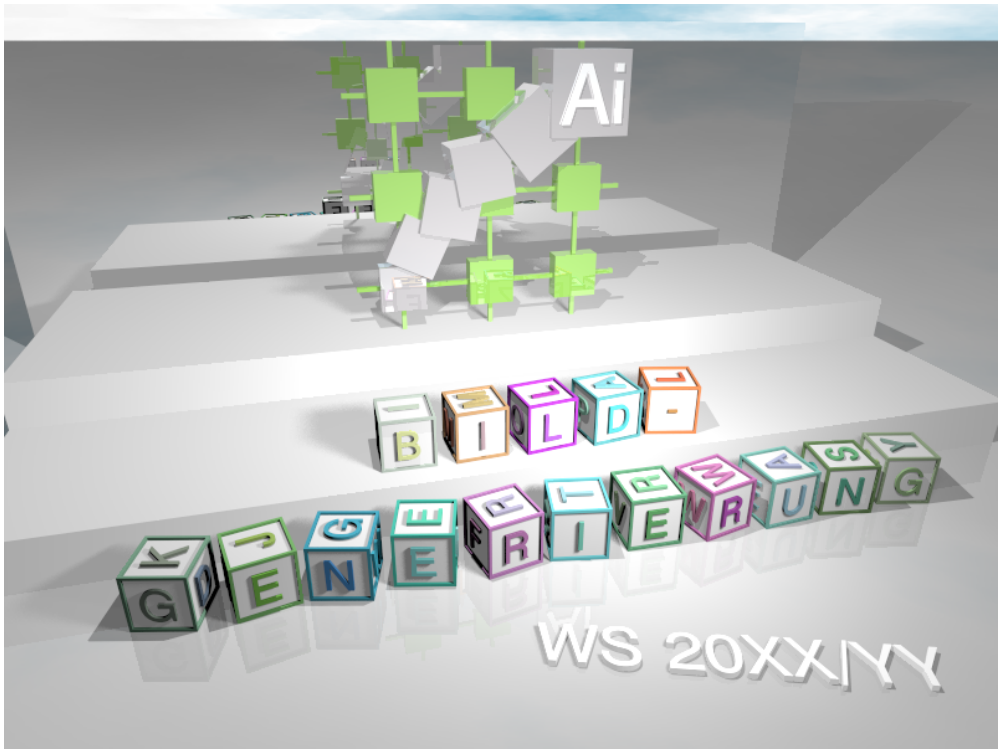
Hinweise:

- Im Verzeichnis `/home/bildgen/Aufgaben/polygone-3` auf dem CIP-Cluster finden Sie ein Rahmenprogramm und Beispiel-Eingabedateien.
- Schreiben Sie zunächst eine Funktion, welche das Polygon mit einer beliebigen Farbe füllt und modifizieren Sie diese anschließend, so dass mit einem Muster gefüllt wird.
- Durchlaufen Sie die Zeilen y_{min} bis y_{max}, die das Polygon enthalten, und datieren Sie in jedem Schritt eine *Tabelle der aktiven Kanten* auf. Es ist hilfreich, dies als eine Liste zu implementieren, die zu jeder Kante, welche die aktuelle y-Scan-Line schneidet, deren oberen Endpunkt, den aktuellen x-Wert sowie die Steigung enthält. Die Kanten werden nach x und ggf. nach Steigung sortiert gespeichert.

- Sorgen Sie dafür, dass die Füllung translationsinvariant, also unabhängig davon ist, an welcher Stelle des Bildes sich das Polygon befindet.
- Der Befehl `pic.drawPoint(x, y, QColor(muster.pixel(xmuster,ymuster)))` malt im Bild `pic` an die Stelle (`x`, `y`) einen Punkt in der Farbe, die das Muster `muster` an der Stelle (`xmuster`, `ymuster`) enthält. Breite und Höhe des Musters können Sie mit `muster.width()` und `muster.height()` ermitteln.

Aufgabe 26 (Modellierung mit Povray I, 5 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/povray-1` finden Sie die Datei `modellierung.pov`. Erweitern Sie dieses Povray-Skript an den markierten Stellen, damit Sie etwa folgendes Bild erhalten:



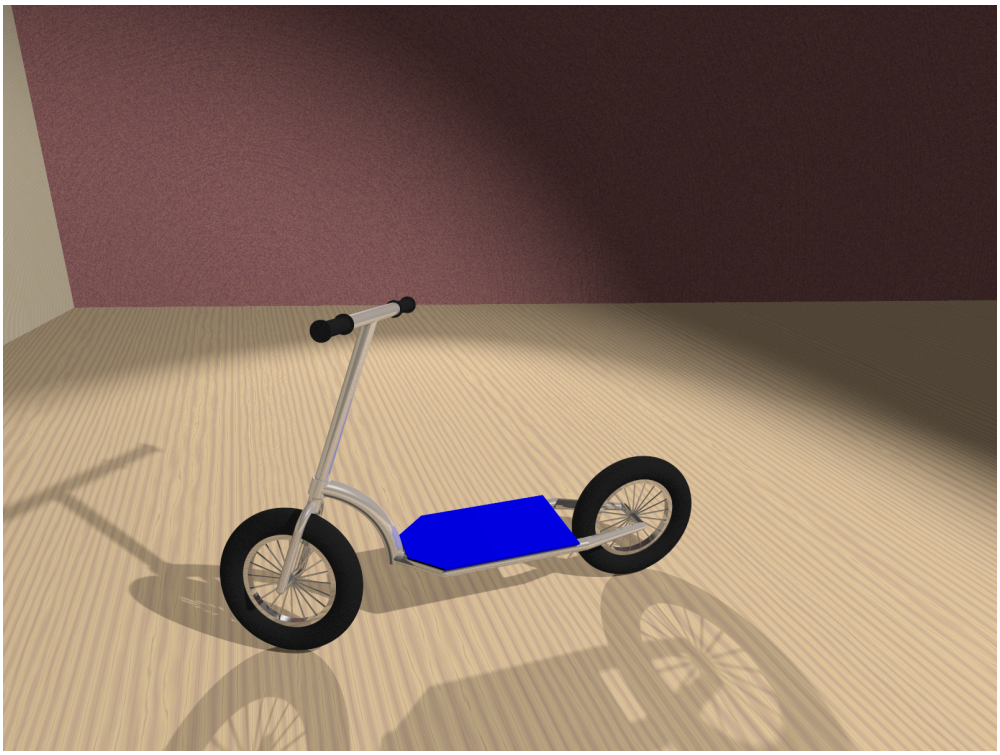
Sie müssen

- Lichtquellen,
- das grüne Gitter des Logos der Arbeitsgruppe „Angewandte Informatik“,
- die fehlenden Würfel des Textes „Bildgenerierung“,
- einen Teil des Podestes und
- den Schriftzug „WS 2021/22“

ergänzen.

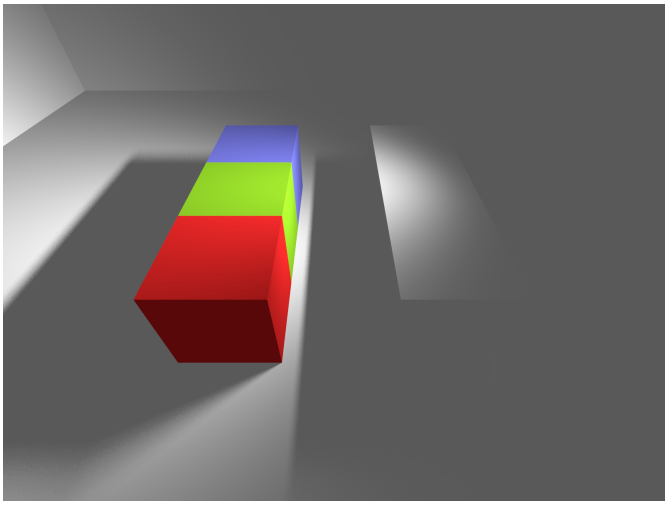
Aufgabe 27 (Modellierung mit Povray II, 5 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/povray-2` finden Sie die Dateien `roller.pov` und `roller.inc`. Erweitern Sie die Datei `roller.inc` an den markierten Stellen, damit Sie etwa folgendes Bild erhalten:



Aufgabe 28 (Radiosity mit Povray, 2 Punkte)

Im Verzeichnis `/home/bildgen/Aufgaben/povray-3` finden Sie die Datei `radiosity.pov`. Erweitern Sie das Povray-Skript an den markierten Stellen. Aktivieren Sie die Radiosity-Implementierung Povrays durch Setzen von „useRadiosity“. Sie sollten nun farbige Gegenstände so platzieren, dass Sie auf dem weißen Quader ein „Abfärben“ erkennen können. Das folgende Bild zeigt ein Beispiel für die Platzierung der Objekte, in dem allerdings Radiosity noch nicht aktiviert ist:



Hinweise zu beiden Aufgaben:

- Es wird jeweils ein `Makefile` mitgeliefert, mit dessen Hilfe Sie die Bilder einfach per Eingabe von `make` erstellen können.
- Höhere Auflösungen erhalten Sie mit `make SIZE=xga` oder `make SIZE=full` – evtl. nach `make clean`.

- Schneller geht es mit `make MODE=quick` – aber die Qualität wird schlechter.
- Einige einfache Beispielskripte sind beigefügt.
- Auf der Webseite http://www.povray.org/documentation/current/ finden Sie ein Tutorium und eine Befehlsreferenz.

Aufgabe 29 (Perspektivische Projektion, 10 Punkte)

Programmieren Sie die perspektivische Projektion, die Überführung in normalisierte Koordinaten sowie die Umwandlung in Gerätekoordinaten. Ergänzen Sie hierzu im Rahmenprogramm `proj1.cc` im Verzeichnis `/home/bildgen/Aufgaben/projektion-1` die entsprechenden Teile der Funktionen

```
Matrix4x4 berechneTransformation(const Vec3D& cop, const Vec3D& vrp,
                                 const Vec3D& vup, int w, int h,
                                 double& un, double& vn)
```

und

```
void maleLinien(Drawing& pic, const vector<Kante>& kanten,
                const Matrix4x4& t, double un, double vn)
```

Alle für die Berechnung benötigten Operationen sind bereits vorgegeben, Sie können also z. B. die Matrix-Multiplikation direkt mittels *-Operator verwenden. Unterstützt werden ausschließlich 4×4-Matrizen, geben Sie deshalb bei der Projektion und den nachfolgenden Transformationen für die n-Koordinate eine Nullzeile in der Matrix an.

Gehen Sie für die Berechnung der Transformation in `berechneTransformation()` in den folgenden Schritten vor und erzeugen Sie zunächst jeweils einzelne Matrizen mit dem gewünschten Effekt. `vpn`, `umin`, `umax`, `vmin` und `vmax` sind bereits vorgegeben.

1. Verschiebung des Aufpunktes der Projektionsebene (`vrp`) in den Ursprung.
2. Überführung in das (u, v, n)-Koordinatensystem.
   a) Bestimmung des (u, v, n)-Koordinatensystems aus der Normalen der Projektionsebene, `vpn`, und der Aufwärtsrichtung, `vup`.
   b) Rotation $z \mapsto n$, $x \mapsto u$, $y \mapsto v$.
3. Verschieben der transformierten Augenposition (`cop` überführt in das (u, v, n)-Koordinatensystem) in den Koordinatenursprung.
4. Standard perspektivische Projektion auf die (u, v)-Ebene.
5. Normalisierung der Koordinaten.
   a) Translation der unteren linken Ecke des Projektionsfensters in den Ursprung.
   b) Skalierung des Projektionsfensters, so dass es in das Einheitsquadrat passt.
   c) Speichern der Ausdehnung des Bildes im Einheitsquadrat, u_n und v_n (der entsprechende Code-Abschnitt ist vorgegeben).

Berechnen Sie nun die Gesamttransformation durch Matrix-Multiplikation in der richtigen Reihenfolge. Die Skalierung auf Gerätekoordinaten ist nicht Teil der Matrix, sie wird im Folgenden separat durchgeführt. Praktischer Grund hierfür ist das einfache 3D-Clipping in normalisierten Koordinaten.

Zum Zeichnen des Bildes ist dann noch Folgendes in `maleLinien()` zu tun:

1. Anwenden der Transformationsmatrix auf Anfangs- und Endpunkt der einzelnen Linien (der entsprechende Code-Abschnitt ist vorgegeben).
2. Umwandlung der homogenen Koordinaten in 2D-Koordinaten.
3. Skalierung auf Fenstergröße (Gerätekoordinaten) unter Verwendung von u_n und v_n.

Testen Sie Ihre Transformation mit den `*.in`-Dateien, z. B. „`proj1 < Colosseum.in`“.

Aufgabe 30 (3D-Clipping für perspektivische Projektion, 12 Punkte)

Die Projektion aus Aufgabe 29 enthielt noch einige Vereinfachungen. So wurde die z-Koordinate vernachlässigt und kein Clipping in normalisierten Sichtkoordinaten durchgeführt.

Das Rahmenprogramm `proj2.cc` im Verzeichnis `/home/bildgen/Aufgaben/projektion-2` enthält alle Schritte für den allgemeinen Fall. Ihre Aufgabe ist, das fehlende Clipping zu ergänzen. Sie sollen hierbei keine optimierten Verfahren wie Cohen-Sutherland oder Cyrus-Beck-Liang-Barsky auf den 3D-Fall übertragen. Gehen Sie einfach von den Parameterdarstellungen der Gerade aus, d. h.

$$x(t) = x_0 + t(x_1 - x_0), \quad y(t) = y_0 + t(y_1 - y_0), \quad z(t) = z_0 + t(z_1 - z_0), \quad 0 \le t \le 1\,.$$

Alle notwendigen Parameter und Daten sind im Code vorgegeben. Für jede Kante wird die Funktion `clip3D` aufgerufen. Die Endpunkte der unbearbeiteten Kante werden als Referenz via `anf` und `end` übergeben. Führen Sie nun für die jeweilige Kante *nacheinander* Clipping an den Clippingebenen in normalisierten Sichtkoordinaten aus. Aktualisieren Sie dann `anf` und `end` mit den neuen Endpunkten der geclippten Kante.

1. $-1 \le z \le z_{\min}$

   Bestimmen Sie die Schnittpositionen t_1, t_2 mit der Front- und Backplane aus $z(t_1) = -1$ und $z(t_2) = z_{\min}$. Beachten Sie auch den Sonderfall für Geraden, die in der x-y-Ebene liegen.

2. $z \le x \le -z$

   Bestimmen Sie t_1, t_2 aus $x(t_1) = z(t_1)$ und $x(t_2) = -z(t_2)$.

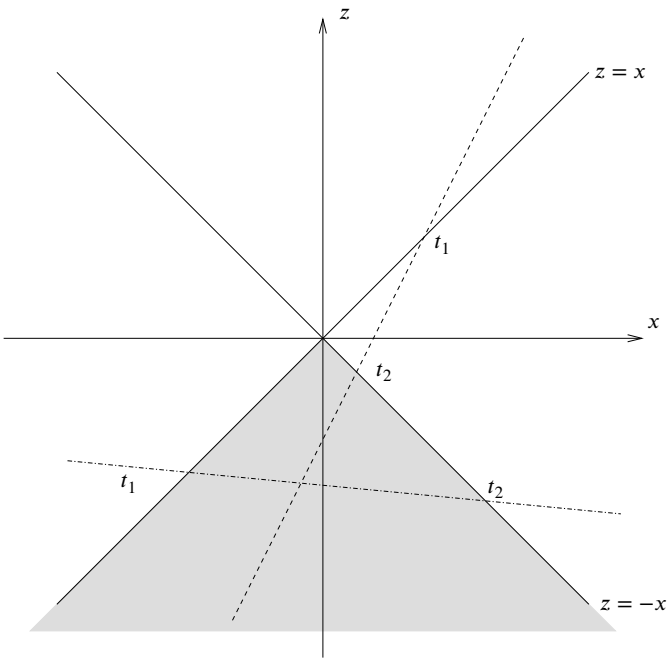
   Überlegen Sie sich, welche verschiedenen Lagen eine Kante und ihre Endpunkte haben kann, so dass sie die zulässige (graue) Fläche schneidet (siehe Illustration). Dokumentieren Sie die verschiedenen Fälle entweder in Ihrem Quelltext oder mit Hilfe separater Zeichnungen (oder beides). Beachten Sie hier den Sonderfall, dass die Gerade parallel zur y-Achse liegt.

3. $z \le y \le -z$

   Analog zu Fall 2.

Testen Sie das Programm mit den Dateien `Wuerfel?.info` (es ist jeweils kommentiert, welcher Teil ausgeblendet werden sollte), `Colosseum?.info` und `sts1.info` sowie den zugehörigen Modellen `Wuerfel.in`, `Colosseum.in` und `sts.in`. Hierzu müssen Sie Ihr Programm z.B. wie folgt aufrufen:

```
cat Wuerfel.in Wuerfel1.info | proj2
```



Aufgabe 31 (z-Buffer-Verfahren, 4 Punkte)

In der Datei `proj3.cc` unter `/home/bildgen/Aufgaben/projektion-3` finden Sie ein fast fertiges Programm, das die Lösung zu Aufgabe 29 um das z-Buffer-Verfahren erweitert. Ergänzen Sie die folgenden Funktionen:

- `clip3DPoint`: Im Gegensatz zu Aufgabe 30 wird hier keine Linie gekappt, sondern lediglich getestet, ob ein gegebener Punkt im kanonischen Bildraum liegt.
- `drawPointZ`: Malt einzelne Pixel und aktualisiert den z-Buffer.

Aufgabe 32 (Färbung – Beleuchtungsmodell nach Phong, 4 Punkte)

Das Rahmenprogramm `proj4.cc` im Verzeichnis `/home/bildgen/Aufgaben/projektion-4` implementiert die perspektivische Projektion samt Füllung der Dreiecksflächen. Ergänzen Sie das Programm um eine pro Gitterpunkt berechnete Beleuchtung. Implementieren Sie hierzu in der Funktion

```
VecRGB berechneBeleuchtung(const Vec3D& ecke, const Vec3D& normale,
                           const Vec3D& auge, const Vec3D& licht,
                           const DrawColour& farbe)
```

das Beleuchtungsmodell nach Phong für *eine* Lichtquelle. Die Funktion wird für jeden der drei Eckpunkte einer Dreiecksfläche aufgerufen und soll die Farbe an diesem Punkt berechnen. Die Interpolation der Farbe über die Fläche (Gouraud-Shading) übernimmt das Rahmenprogramm für Sie.

Die folgenden Größen sind vorgegeben:

`Vec3D ecke`	Position der Ecke in Weltkoordinaten
`Vec3D normale`	Normale zur Fläche in dieser Ecke (in Weltkoordinaten)
`Vec3D auge`	Koordinaten des Auges in Weltkoordinaten
`Vec3D licht`	Koordinaten der Lichtquelle in Weltkoordinaten
`VecRGB lightAmbient`	ambiente Lichtintensität
`VecRGB lightDiffuse`	diffuse Lichtintensität
`VecRGB lightSpecular`	Lichtintensität für winkelabhängige Reflexion
`VecRGB materialAmbient`	ambienter Reflexionskoeffizient des Materials
`VecRGB materialDiffuse`	diffuser Reflexionskoeffizient des Materials
`VecRGB materialSpecular`	winkelabhängiger Reflexionskoeffizient des Materials
`double materialSpecularity`	Exponent für winkelabhängige Reflexion
`double c0, c1, c2`	Konstanten für entfernungsabhängige Dämpfung

Die Rechnung mit Farbwerten innerhalb der Funktion erfolgt über dreidimensionale Vektoren, deren Einträge die drei Kanäle des RGB-Farbmodells darstellen. Die RGB-Intensitäten sind auf $[0, 1]$ normiert. Für Lichter stellen diese Werte die Lichtintensität des jeweiligen Farbkanals dar, für Materialien bzw. Objektoberflächen deren Reflexionskoeffizienten für den jeweiligen Farbkanal.

Um nicht jeden Farbkanal einzeln berechnen zu müssen sind neben `+` und `-` geeignete Operatoren vorgegeben, z. B. `VecRGB v1 * VecRGB v2` für elementweise Multiplikation.

Beachten Sie, dass negative Skalarprodukte auf dem Auge bzw. der Lichtquelle abgewandte Flächen hindeuten. Im Falle eines negativen Skalarproduktes ist der entsprechende Lichtanteil auf Null zu setzen, um keine negativen Lichtintensitäten zu erzeugen. Hierzu können sie die `max()`-Funktion benutzen.

Testen Sie Ihr Programm mit den `*.in`-Dateien.

Aufgabe 33 (Parallelprojektion, 6 Punkte)

Gegeben seien die folgenden Größen:

$$VRP = \begin{pmatrix} 3 \\ 1 \\ 10 \end{pmatrix}, COP = \begin{pmatrix} 4 \\ 2 \\ 15 \end{pmatrix}, VPN = \begin{pmatrix} 2 \\ 3 \\ 2 \end{pmatrix}, VUP = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix},$$

$$u_{\min} = 0, u_{\max} = 40, v_{\min} = 0, v_{\max} = 30, B = 0, F = 10.$$

Berechnen Sie die fünf für die Parallelprojektion benötigten Matrizen $T(-VRP)$, R, SH_{par}, T_{par} und S_{par} und bestimmen Sie damit die Projektion des Punktes

$$Q = \begin{pmatrix} 10 \\ 10 \\ -10 \end{pmatrix}$$

in den kanonischen Bildraum.

Aufgabe 34 (Rundflug um den Eiffelturm, 2 Punkte)

Erzeugen Sie eine Animation, die einen spiralförmigen Rundflug um den Eiffelturm zeigt. Fahren Sie hierzu nicht nach Paris, sondern verwenden Sie die Daten aus der Datei `toureiffel.in`. Starten Sie Ihren Flug am Boden ($y = -662$) und fliegen Sie bis zur Spitze ($y = 550$), wobei Sie den Turm einmal umrunden. Halten Sie einen konstanten Abstand von 800 Längeneinheiten von der Mitte des Turms (y-Achse) und blicken Sie immer waagerecht zur Mitte. Dann passen Ihre Aufnahmen auf Bilder der Größe 250×400 Pixel. Der Abstand zur Projektionsebene spielt hier keine große

Rolle und kann recht willkürlich gewählt werden. Setzen Sie Ihren Film aus 100 Einzelbildern zusammen.

Verwenden Sie die Eingabedatei `toureiffel.in` und ignorieren Sie die Werte der ersten beiden Zeilen (COP und TGT). Erzeugen Sie aus Ihrem Flug eine animierte `gif`- oder `mpg`-Datei. Ein Beispielprogramm zur Erzeugung von Animationen finden Sie im Verzeichnis `/home/bildgen/Aufgaben/rundflug`. Zum Betrachten der Dateien können Sie `gwenview` verwenden, für `gif`-Ausgaben zusätzlich auch `eog`.

Falls Sie Aufgabe 32 nicht bearbeitet haben, lassen Sie die Beleuchtungsfunktion einfach die übergebene Farbe zurückgeben und setzen Sie den `outline` Parameter von `maleDreiecke` auf `true`.

Aufgabe 35 (Weitere Vorschläge, 0 Punkte)

Bauen sie ähnlich wie in `anim01.gif` ein dreidimensionales Nikolaushaus. Oder machen Sie einen Rundflug über das Colosseum …

Aufgabe 36 (Raytracing nach Whitted, 15 Punkte)

Implementieren Sie rekursives Raytracing nach Algorithmus 10.5 und 10.6 des Skripts und verwenden Sie für das Auffinden der vom Strahl getroffenen Objekte den „naiven Ansatz“ von Seite 10-10.

Im Verzeichnis `/home/bildgen/Aufgaben/raytracing-1` finden Sie ein Archiv mit einem Rahmenprogramm, in dem Sie die Implementierung vornehmen können. Sie müssen nur Änderungen an der Datei `raytracer.cc` vornehmen. Compiliert wird das Rahmenprogramm durch Eingabe von `make`.

Folgende Methoden, Objekte und Strukturen werden für die Implementierung benötigt:

- Die Klasse `Vector3d` stellt einen Vektor mit drei `double`-Komponenten dar und besitzt unter anderem die Methoden
  - `cross(const Vector3d&)` zur Berechnung des Kreuzproduktes mit einem weiteren Vektor,
  - `dot(const Vector3d&)` zur Berechnung des Innenproduktes mit einem weiteren Vektor,
  - `norm()` liefert die euklidische Norm des Vektors und
  - `cwiseProduct(const Vector3d&)` kann zur komponentenweisen Multiplikation mit einem weiteren Vektor verwendet werden.

  Außerdem sind die Operatoren `+`, `-`, `+=` sowie `-=` zur Addition und Subtraktion von Vektoren, die Operatoren `*` und `*=` zur Multiplikation mit Skalaren und die Operatoren `/` und `/=` zur Division von Vektoren durch Skalare überladen.
- Die Klasse `Image` enthält das erzeugte Bild und stellt die Methoden
  - `getWidth()` und `getHeight()`
  - sowie `setPixel(int x, int y, const Vector3d &rgb)`

  bereit. Verwenden Sie die Methode `Raytracer::clampCol(Vector3d&)` vor dem Aufruf von `setPixel`, damit die Farbwerte auf das Intervall [0; 1] beschränkt werden.
- Die Klasse `Scene` enthält sämtliche die Szene betreffenden Informationen. Sie benötigen folgende Methoden und Members:
  - `getCamera()` liefert ein Objekt der Klasse `Camera`, die weiter unten erläutert wird.

  - `objects` ist ein Vektor, der Pointer auf alle in der Szene vorhanden Objekte vorhält.
  - `getBackground()` liefert die Hintergrundfarbe der Szene.
  - `lights` ist ein Vektor, der Pointer auf alle in der Szene vorhanden Lichtquellen vorhält.
- Sie benötigen folgende Methoden der Klasse `Camera`:
  - `getPosition()` liefert die Position der Kamera als `Vector3d`.
  - `getDirection()` liefert die Blickrichtung der Kamera als `Vector3d`.
  - `getUp()` liefert den Up-Vektor der Kamera als `Vector3d`.
  - `getHorAngle()` liefert den horizontalen Blickwinkel der Kamera als `double`.
- Die verschiedenen Objekt-Typen sind von der Klasse `Object` abgeleitet und Sie benötigen:
  - `isHitBy(const Ray&)` gibt eine Instanz der Struktur `HitInfo` zurück.
- Alle Lichtquellen sind Instanzen der Klasse `Light` und Sie benötigen:
  - `visibleFrom(const Scene&, const Vector3d&)` bestimmt, ob die Lichtquelle vom übergebenen Punkt aus sichtbar ist und gibt die Information in einer Instanz der Struktur `VisibleInfo` zurück.
- Die Struktur `Ray` stellt einen Strahl mit Ausgangspunkt und Richtung dar. Alle Informationen dazu finden Sie in der Datei `ray.h`.
- Die Struktur `HitInfo` bündelt die Schnittpunkt-Informationen eines Strahls mit einem Objekt. Alle Informationen dazu finden Sie in der Datei `hitinfo.h`.
- Die Struktur `VisibleInfo` bündelt die Informationen, ob eine Lichtquelle von einem Punkt aus sichtbar ist. Alle Informationen dazu finden Sie in der Datei `visibleinfo.h`.

Aufgabe 37 (Raytracing nach Whitted - Teil 2, 8 Punkte)

Erweitern Sie die Implementierung zum rekursiven Raytracing aus Aufgabe 36 um Lichtbrechung und die Darstellung von Tetraedern.

Im Verzeichnis `/home/bildgen/Aufgaben/raytracing-2` finden Sie ein ein Rahmenprogramm, in dem Sie die Implementierung vornehmen können. Sie müssen nur Änderungen an den markierten Stellen in den Dateien `raytracer.cc`, `triangle.cc` und `tetraeder.cc` vornehmen. Orientieren Sie sich für die Implementierung der Klassen `Tetraeder` und `Triangle` an den Klassen `Box` und `Parallelogram`. In der Datei `README` wird beschrieben, wie Sie das Rahmenprogramm compilieren können.

Zusätzlich zu den erläuterten Methoden, Objekten und Strukturen aus Aufgabe 36 benötigen Sie die Methoden

- `Object::getRefraction()`, die den Brechungsindex für das Medium des Objekts als `double` liefert, und
- `Object::getPigment()`, die einen `Vector3d` zurückgibt und die Durchlässigkeit des Objekts für die drei Farbkomponenten darstellt.

Aufgabe 38 (Silhouetten-Algorithmus, 10 Punkte)

Implementieren Sie den Silhouetten-Algorithmus und stellen Sie wie in Aufgabe 21 mit dessen Hilfe den Graphen der Funktion f dar.

Verwenden Sie die gleiche Parallelprojektion wie in Aufgabe 21.

Das Rahmenprogramm `silhouetten.cc` im Verzeichnis `/home/bildgen/Aufgaben/silhouetten` dient als Grundlage. Implementieren Sie darin die Funktionen

```
Matrix4x4 berechneMpar(double umin, double umax, double vmin, double vmax,
                       double& ratio)
void erzeugeKurven(double xmin, double xmax, double zmin, double zmax,
                   int num, int pieces,
                   const std::function<double(double,double)>& func,
                   std::vector<std::vector<Vec3D>>& kurven )
void maleSilhouetten(Drawing& pic,
                     const std::vector<std::vector<Vec3D>>& kurven,
                     const Matrix4x4& mpar, double ratio)
```

bzw. die fehlenden Teile der Funktionen. Weitere Informationen zu den Ein- und Ausgabeparametern finden Sie im Rahmenprogramm.

Aufgabe 39 (B-Splines, 3 Punkte)

Schreiben Sie eine Funktion zum Zeichnen von B-Splines. Ergänzen Sie hierzu in Ihrer Lösung zu Aufgabe 40 die Funktion

```
void maleBSpline(Drawing& pic, const vector<DPoint2D>& p, int n)
```

die den zu $p_0, \ldots, p_m$ gehörenden B-Spline (in einer anderen Farbe) malt. Markieren Sie zusätzlich zum Zeichnen der $m - 2$ Abschnitte die Knotenpunkte zwischen diesen.

Aufgabe 40 (Bézier-Kurven, 3 Punkte)

Schreiben Sie eine Funktion zum Zeichnen von Bézier-Kurven. Ergänzen Sie hierzu das Rahmenprogramm `curves.cc` im Verzeichnis `/home/bildgen/Aufgaben/splines-2` entsprechend.

Es seien $m + 1$ Punkte $p_0, \ldots, p_m$ gegeben. Für die Funktion

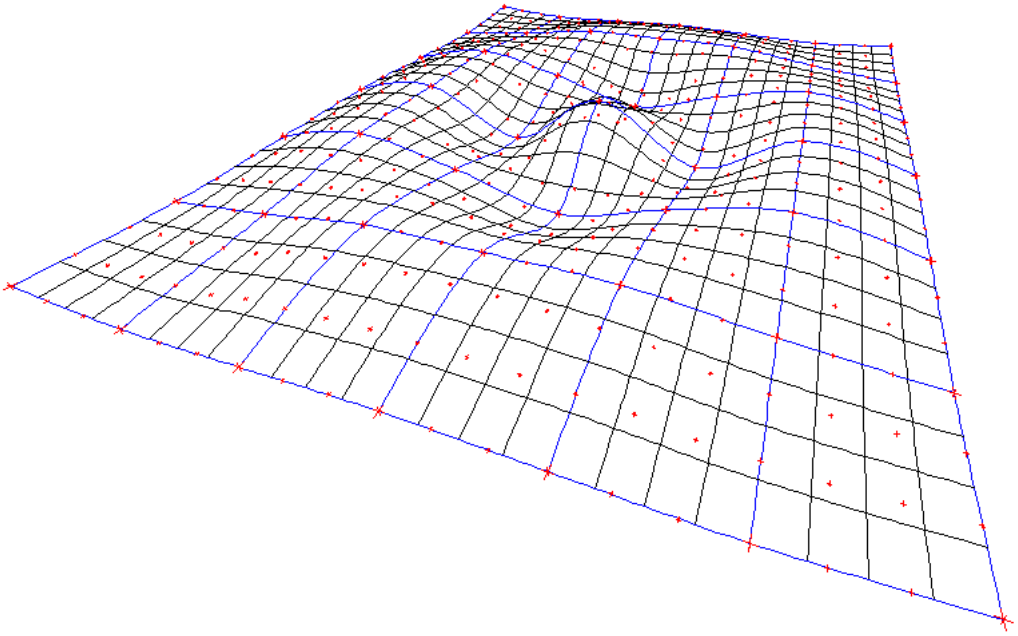
```
void maleBezierKurve(Drawing& pic, const vector<DPoint2D>& p, int n)
```

ist vorausgesetzt, dass m ein Vielfaches von 3 ist. Die Kurve besteht dann aus $\frac{m}{3}$ einzelnen Kurvenstücken.

Sie können zum Testen die Dateien `points?.in` benutzen.

Aufgabe 41 (Bézier-Flächen, 10 Punkte)

Ergänzen Sie eine Funktion `berechneBezierFlaeche`, welche die Kanten der Bézier-Fläche bestimmt, im Rahmenprogramm `surfaces.cc` unter `/home/bildgen/Aufgaben/flaechen`.



Zu jeweils 16 Punkten `p[i][j]`, ..., `p[i + 3][j + 3]` gehört ein Flächenstück bestehend aus `anzkurv` Kurvenstücken für jede der beiden Richtungen, wobei jedes Kurvenstück durch `anzlin` Linien approximiert wird.

Mit dem Befehl `vk.push_back(Kante(anf, end, BLACK));` können Sie dem Kanten-Vektor die einzelnen Kantenstücke hinzufügen, wobei `anf` und `end` vom Typ `Vec4D` sind.

Aufgabe 42 (Catmull-Rom-Splines, 8 Punkte)

Die Catmull-Rom-Splines können durch die Basismatrix

$$M_{CR} = \frac{1}{2}\begin{pmatrix} -1 & 2 & -1 & 0 \\ 3 & -5 & 0 & 2 \\ -3 & 4 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix}$$

definiert werden. Hierbei handelt es sich um einen interpolierenden Spline, d. h. zu gegebenen Punkten $p_0, \ldots, p_m$ werden $p_1, \ldots, p_{m-1}$ durch eine Kurve verbunden.

1. Zeigen Sie, dass die Kurve wirklich durch $p_1, \ldots, p_{m-1}$ verläuft und damit die Stetigkeit.
2. Berechnen Sie die Tangenten in den Punkten p_i, $i = 1, \ldots, m-1$, und zeigen Sie so, dass die Kurve C^1-stetig ist.
3. Ist die Kurve C^2-stetig? (Begründung!)
4. Ergänzen Sie in Ihrer Lösung zu Aufgabe 40 die Funktion `maleCRSpline`.

Aufgabe 43 (Hermite-Kurven, 5 Punkte)

Ergänzen Sie im Rahmenprogramm `hermite.cc`, welches Sie im Verzeichnis `/home/bildgen/Aufgaben/splines-1` finden können, die Funktion

```
void maleHermiteKurve( Drawing& pic, DPoint2D p1, DPoint2D p4,
                       DPoint2D r1, DPoint2D r4, int n )
```

die eine Hermite-Kurve zwischen zwei Punkten $p_1, p_4 \in \mathbb{R}^2$ mit Tangentenvektoren $r_1, r_4 \in \mathbb{R}^2$ zeichnet. Approximieren Sie die Kurve durch eine Folge von n Linien, indem Sie $n-1$ Zwischenpunkte bestimmen.

Aufgabe 44 (Hermite-Kurven Animation, 4 Punkte)

Verwenden Sie Ihre Lösung aus Aufgabe 43, um sich die Abhängigkeit der Kurven von den Tangentenrichtungen und -längen in einer Animation zu veranschaulichen.

a) Starten Sie mit einer Geraden und lassen Sie dann den Tangentenvektor r_4 um den Endpunkt p_4 rotieren.

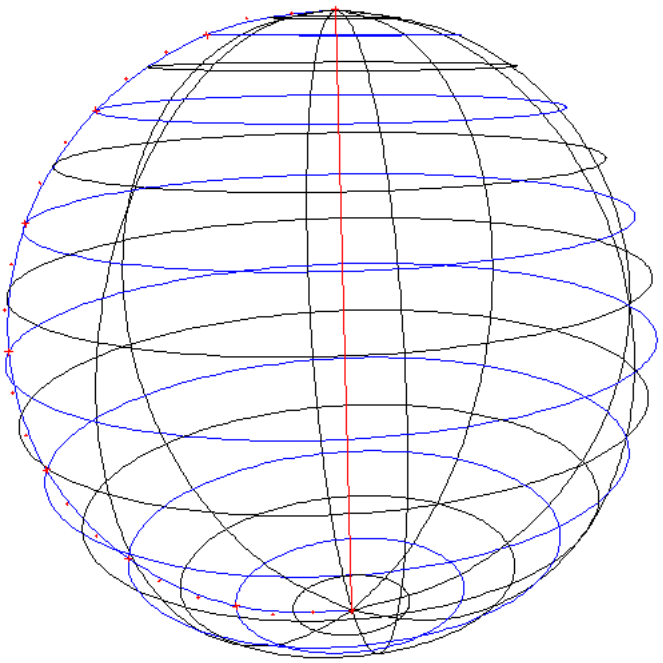
b) Legen Sie die beiden Endpunkte auf die Gerade $y = x$, verwenden Sie im unteren Punkt einen Tangentenvektor nach rechts (links) und im oberen Punkt einen Tangentenvektor nach oben (unten). Variieren Sie die Längen der Tangentenvektoren.

Aufgabe 45 (Hermite-Kurven Schleife, 4 Punkte)

Betrachten Sie den Fall $p_1 = (a, a)$, $p_4 = (a + b, a)$, $r_1 = (\rho, \rho)$, $r_4 = (\rho, -\rho)$ für $\rho \in \mathbb{R}$ und festes $a, b \in \mathbb{R}$. Skizzieren Sie, welche Kurventypen sich für verschiedene Werte von ρ ergeben und berechnen Sie, ab wann eine Schleife entsteht.

Aufgabe 46 (Rotationskörper, 10 Punkte)

Ergänzen Sie die Funktion `berechneRotationsKoerper`, welche die Kanten der (vertikalen) Bézier-Splines und die Kanten der (horizontalen) Kreise des Rotationskörpers bestimmt, im Rahmenprogramm `rotk.cc` unter `/home/bildgen/Aufgaben/rotationskoerper`.



Zu jeweils 4 Punkten `p[i]`, ..., `p[i + 3]` in der x-y-Ebene gehören

- `anzkurv` Bézierkurven, die durch `anzlinku` Geradenstücke approximiert werden und durch Drehung der „Originalkurve“ um die x-Achse entstehen,
- `anzkreis` Kreise um die x-Achse, die durch `anzlinkr` Geradenstücke angenähert werden.

Beachten Sie, dass die x-Achse hier ausnahmsweise nach oben zeigt. Wie in Aufgabe 41 sollen alle Geradenstücke in einen Kantenvektor eingefügt werden.

Aufgabe 47 (2D-Transformation, 4 Punkte)

Gegeben seien die folgenden Punkte im $\mathbb{R}^2$:

$$P_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix},\ P_2 = \begin{pmatrix} 1 \\ 4 \end{pmatrix},\ P_3 = \begin{pmatrix} -1 \\ 2 \end{pmatrix},\ Q_1 = \begin{pmatrix} 4 \\ -10 \end{pmatrix},\ Q_2 = \begin{pmatrix} -7 \\ -3 \end{pmatrix},\ Q_3 = \begin{pmatrix} -5 \\ 3 \end{pmatrix}$$

Berechnen Sie die Transformationsmatrix $T \in \mathbb{R}^{3\times3}$, die (nach Übergang auf homogene Koordinaten) die Punkte P_i auf die Punkte Q_i für $i = 1, \ldots, 3$ abbildet.

Aufgabe 48 (Transformationen in der Ebene, 6 Punkte)

Durch drei (nicht-kollineare) Punkte p_1, p_2, p_3, $p_i = \begin{pmatrix} p_{i,x} \\ p_{i,y} \end{pmatrix}$, und drei Bildpunkte q_1, q_2, q_3 ist eine Transformation t eindeutig bestimmt. Verwendet man homogene Koordinaten, so gilt:

$$\begin{pmatrix} q_{i,x} \\ q_{i,y} \\ 1 \end{pmatrix} = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_{i,x} \\ p_{i,y} \\ 1 \end{pmatrix}$$

Ergänzen Sie in der Datei `transfo_rahmen.cc` die Funktionen zur Berechnung und Anwendung einer Transformation.

Aufgabe 49 (Spiegelung an einer Ebene im Raum, 5 Punkte)

Eine Ebene im $\mathbb{R}^3$ ist bestimmt durch einen Normalenvektor $n = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \in \mathbb{R}^3$, $\|n\|_2 = 1$ und einen Abstand $c \in \mathbb{R}$ vom Ursprung. Für einen Punkt $r = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ der Ebene gilt: $n^T r - c = 0$.

Bestimmen Sie die Transformationsmatrix $T \in \mathbb{R}^{4\times4}$, welche die Spiegelung an dieser Ebene beschreibt.

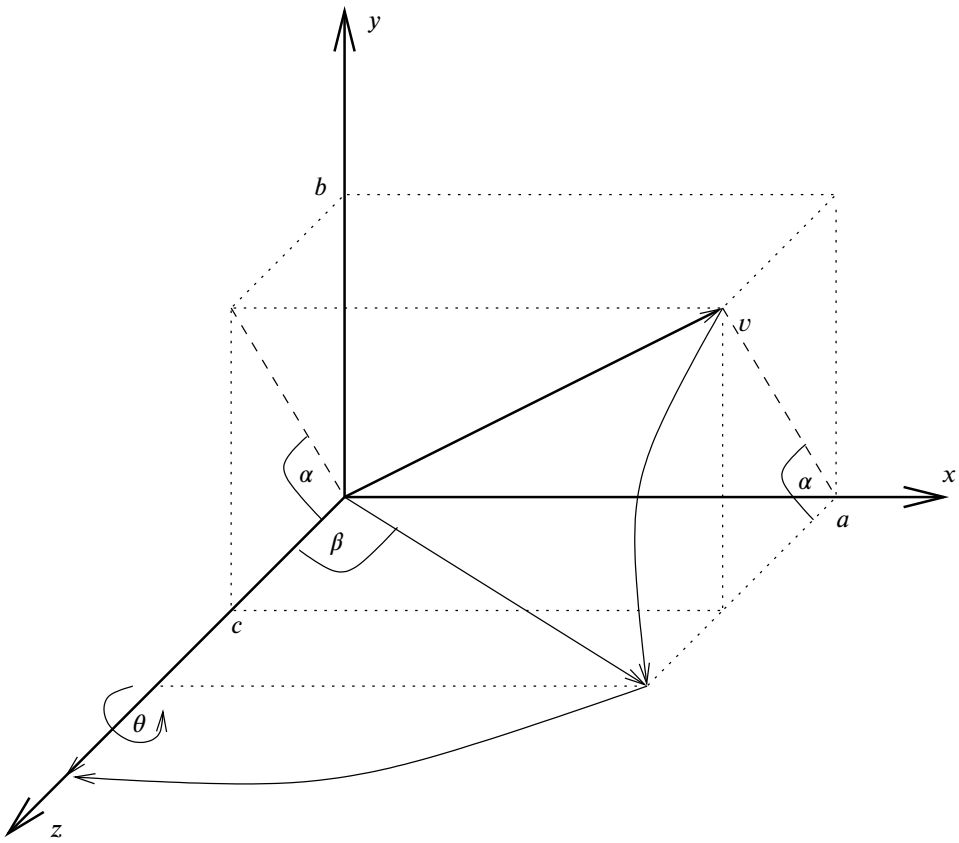
Aufgabe 50 (Punktspiegelung im Raum, 3 Punkte)

Gegeben sei ein Punkt $S = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \in \mathbb{R}^3$.

Bestimmen Sie die Transformationsmatrix $T \in \mathbb{R}^{4\times4}$, die die Punktspiegelung an S in homogenen Koordinaten beschreibt.

Aufgabe 51 (Drehung um eine Gerade durch den Nullpunkt, 10 Punkte)

Gegeben sei ein Richtungsvektor $v = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \in \mathbb{R}^3$, $\|v\|_2 = 1$, und ein Winkel θ. Bestimmen Sie die Transformationsmatrix für die Drehung um den Winkel θ um die Gerade, die durch den Ursprung in Richtung von v verläuft.



Bringen Sie v durch Rotation um die x- und y-Achse zuerst in Richtung der z-Achse, rotieren Sie dort um θ, und machen Sie anschließend die ersten beiden Rotationen rückgängig.