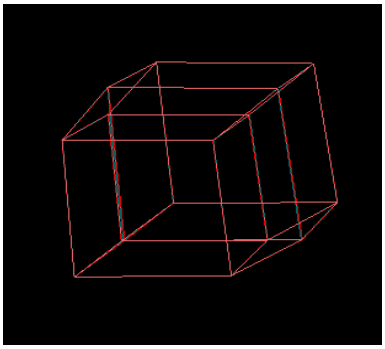
Softwaretechnik-Projekt – Bild & Grafik



Stereoskopie in OpenGL

Simon Heuser
Technische Hochschule Mittelhessen

SS 2017
1. Mai 2018

Inhaltsverzeichnis

Zusammenfassung

Dieser Bericht fasst die Schritte zusammen, die zum Konstruieren einer binokularen Anwendung vorgenommen werden müssen. Dazu sollen zuerst die Konzepte der Stereoskopie erläutert werden. Es wird daraufhin eine Realisierung in OpenGL und GLUT vorgestellt.

Im ersten Kapitel findet eine Erklärung der Stereoskopie und des binokularen Anaglyph-Effekts statt.

Im zweiten Kapitel werden verschiedene Konzepte dargestellt, die ebenfalls in der Anwendung realisiert wurden.

Im dritten Kapitel wird die Konstruktion von vierdimensionalen Objekten erläutert, sowie der Umgang mit diesen in einem dreidimensionalen Koordinatensystem.

Im vierten Kapitel wird das Demoprogramm untersucht, und dabei werden die notwendigen OpenGL- und GLUT-Prozeduren erläutert.

Im fünften Kapitel werden die Erkenntnisse und Probleme, die aus dem Projekt entstanden sind, zusammengefasst.

Weitere Informationen, Details und Hinweise wurden im Anhang gesammelt.

Kapitel 1

Einstieg

1.1 Geschichte der Stereoskopie

Die ursrpünglichen Stereobilder – oder Stereogramme –, die zwischen 1850 und 1930 äußerst beliebt waren, bestanden aus einem Paar nebeneinanderliegender Bilder, die mit einem Stereoskop betrachtet wurden (Abbildung 1.1). Dabei wird den Augen getrennt das jeweilige Bild zugeführt, wodurch der Eindruck eines dreidimensionalen Bildes entsteht.

> "Around the world, independent and entrepreneurial photographers broke into the growing market for illustrations of all types of subjects: local history and events, grand landscapes, foreign monuments, charming genre scenes, portraits of notables and urban architecture. War and disasters such as floods, fires, train-wrecks, and earthquakes were enormously popular subjects." [24]

Die gesellschaftliche Bedeutung der Stereogramme zwischen 1850 und 1910 kann mit der der Television zwischen 1960 und 2010 verglichen werden:

> "During the period between the 1850s and the 1910s, stereos were a mainstay of home entertainment, perhaps second only to reading as a personal leisure activity.
>
> ...
>
> Like cable television in its present diversity and niche marketing, stereos accommodated tastes ranging from vulgar to refined, from simple to scientific. Production quality was also wide-ranging - from exquisitely sharp original silver prints to indistinct, cheaply produced copies, and eventually half-tone photomechanical processes." [24]

Mit dem Wachstum der neuen Medien – Film und Illustrierte – verloren ab 1930 die Stereogramme schließlich an Bedeutung. [24]

Seitdem galt die Beschäftigung mit stereoskopischen Produkten eher als Beschäftigung für interessierte Künstler oder Fotografen.

Als heutige 3D-Techniken sind besonders die Shutter- und Polarisationsbrillen zur Betrachtung von stereoskopischen Kinofilmen zu nennen. Eine günstige und einfache Alternative, die gerne von zahlreichen Stereoskopielaboren und auch in diesem Projekt verwendet wird, ist die Anaglyphenstereoskopie. Dabei wird eine rot-cyanfarbene Brille zur Betrachtung eines Rot-Cyan-Stereobildes (Abbildung 1.2) verwendet.

Abbildung 1.1: Ein Stereogramm, das stereoskopisch betrachtet werden kann [25]

Abbildung 1.2: Ein Anaglyphen-Stereobild [29]

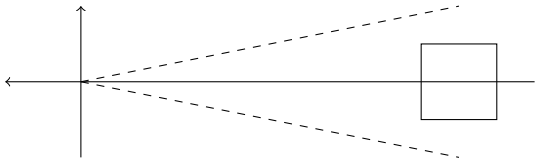


Abbildung 1.3: Monokulare Sicht

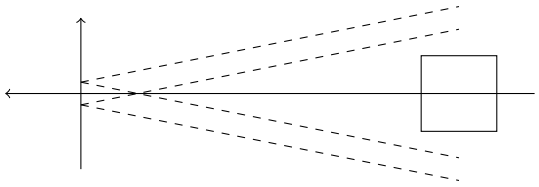


Abbildung 1.4: Binokulare Sicht

1.2 Konzepte der Stereoskopie

Grundsätzlich funktioniert die Illusion des dreidimensionalen Sehens nur, wenn die beiden aus einer Szene erzeugten Abbildungen nebeneinander versetzt dargestellt und – im Falle der Anaglyph-Darstellung – jeweils gegensätzlich eingefärbt werden.

Die versetzte Darstellung ist notwendig, um die naturgegebene *Stereopsis* auszulösen, sodass dem Gehirn eine räumliche Umgebung auf dem tatsächlich flachen Computerbildschirm simuliert wird. Der Vorgang dieser Simulation nennt sich *Stereoskopie*.

1.2.1 Stereopsis

Stereopsis – von gr. *stereós* („fest [Körper]“, „kubisch [Dimension, Zahl]“) und *ópsis* („Sehkraft“, „Erscheinung“) [1] – ist der natürliche Vorgang, beim binokularen Sehen zwei Bilder übereinanderzulagern und dadurch einen dreidimensionalen Eindruck von der Umgebung zu erhalten.

Dieser Vorgang findet ständig und unkontrollierbar statt, kann jedoch leicht überprüft werden, indem man beim Betrachten eines Bildes abwechselnd das linke oder rechte Auge schließt. Intuitiv erkennt man, dass das Gehirn beide Versionen interpoliert und daraus ein passendes Bild erstellt.

Der Grund für die verschobene Wahrnehmung ist natürlich die horizontal versetzte Position der Augen, die in 1.2.2 noch eine Rolle spielen wird. Vom linken Auge aus scheint ein mittig positioniertes Objekt etwas weiter rechts zu sein und vom rechten Auge aus etwas weiter links. Aus dieser *binokularen Disparität* kann das Gehirn die Tiefenverhältnisse der Umgebung ableiten [3].

1.2.2 Stereoskopie

Stereoskopie – von gr. *stereós* („fest [Körper]“, „kubisch [Dimension, Zahl]“) und *skop-éo* („betrachten“, „nachdenken“) [1] – ist der technische Vorgang, eine als dreidimensional wahrnehmbare Abbildung zu erzeugen.

Dabei wird dem linken Auge ein etwas anderes Bild als dem rechten Auge zugeführt, etwa durch farbliches Übertönen beim Anaglyphen – der Gegenstand dieses Demoprogramms – oder abwechselndes gänzliches Verbergen

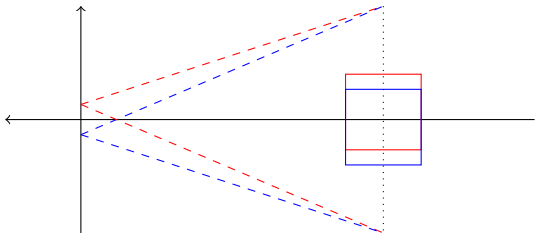


Abbildung 1.5: Binokulare Sicht bei Anaglyphen-Stereoskopie

beim Shutter[1].

Zusätzlich wird für jedes Auge die Sichtachse diametral entlang der x-Koordinate verschoben (*interokulare Distanz*). Dadurch transformiert man die monokulare zur binokularen Sicht auf eine Szene. [2]

Daraufhin muss man noch die *Projektion* der Szene auf einen virtuellen Bildschirm anpassen: die Blickrichtungen des linken und rechten Auges sind parallel zur z-Achse, aber die Abbildungen erscheinen durch die verschiedene Projektion versetzt auf der Bildschirmebene.

1.2.3 Separation

Die Separation ist die Normalisierung des interokularen Abstandes. Die um $\frac{+}{-}\frac{x_{inter}}{2}$ verschobenen Augenpunkte werden durch die Breite des virtuellen Bildschirms dividiert:

$sep = \frac{x_{inter}}{w}$ [2]

Es ist empfehlenswert, die Separation nicht zu hoch zu wählen, damit die beiden Abbildungen nicht zu stark divergieren. [2]

1.2.4 Konvergenz

Dort, wo die Sichtkegel des linken und rechten Auges konvergieren, soll sich der virtuelle Bildschirm befinden. Man nennt diese Entfernung z_{screen}.

Ein Objekt, das auf dem virtuellen Bildschirm liegt (also $z = z_{screen}$), erzeugt keinen binokularen Effekt, denn beide Sichtkegel überlagern sich an dieser Stelle (Abbildung 1.6, außerdem Anhang A.1 für eine Erklärung der Sichtebenen).

1.2.5 Parallaxe

Wenn die Projektionen auf der Betrachtungsebene divergieren, kommt es zur Darstellung der *Parallaxe*. Diese bezeichnet die Entfernung zwischen linkem und rechtem Bild auf dem virtuellen Bildschirm. [2]

$par = x'_R - x'_L$

Bei einem Anaglyph-Programm nimmt man ohne rot-blaue Brille sehr deutlich die Parallaxe zwischen roten und blauen Objekten wahr. Je näher ein Objekt dem Betrachter kommt, desto kleiner wird die Parallaxe; bei einer negativen Parallaxe scheint ein Objekt aus dem Bildschirm zu ragen, bei einer positiven Parallaxe entsteht der

[1] Shutter-Stereoskopie: Ein bewegtes Bild wird mit einer Framerate f angezeigt. Dabei wird abwechselnd ein Frame für das linke und das rechte Auge angezeigt, wobei die Shutterbrille das jeweils inaktive Auge 'verschließt' (das Glas wird schwarz getönt). Daraus ergibt sich für jedes Auge eine Framerate von $\frac{f}{2}$. Wenn diese Frequenz hoch genug ist, ist der Shuttereffekt nicht mehr wahrnehmbar und eine räumliche Wahrnehmung entsteht. [2]

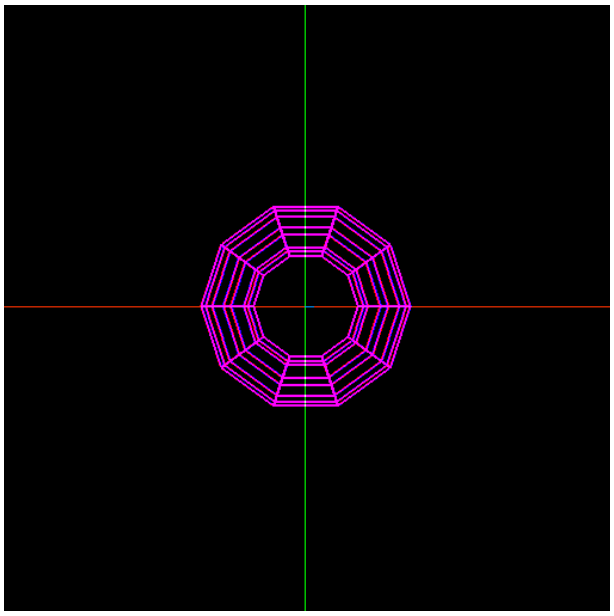


Abbildung 1.6: Darstellung auf der Konvergenzebene [4]

Eindruck, das Objekt verschwinde in den Bildschirm hinein. [2]

Im Beispiel des Anaglyphen heißt das:

- Negative Parallaxe ($x'_L > x'_R$): das rote Bild steht weiter rechts als das blaue Bild – das führt zu einer Illusion von Nähe
- Keine Parallaxe ($x'_L = x'_R$): beide Bilder überlagern sich und sind genau auf der Ebene des virtuellen Bildschirms (Konvergenz)
- Positive Parallaxe ($x'_L < x'_R$): das rote Bild steht weiter links als das blaue Bild – das führt zu einer Illusion von Entfernung (s. Abbildung 1.5)

Dadurch wird schon deutlich, dass die Parallaxe in einer funktionalen Abhängigkeit zur z-Koordinate steht. Die Funktion ist aber nicht, wie man vermuten könnte, linear, sondern verläuft logarithmisch: für kleine z fällt sich die Parallaxe sehr schnell (die Bilder divergieren schneller), während um die Konvergenzebene herum eine Mäßigung der Parallaxe zu beobachten ist. Für große z ist dann bald keine Änderung der Parallaxe mehr erkennbar, bis die Bilder von der Far Clipping Plane 'abgeschnitten' werden. (Abbildung 1.8)

Die Parallaxe-Funktion zeigt auch, dass die Separation die größtmögliche auftretende Parallaxe darstellt. [2]

1.3 Anaglyphen

Anaglyphen – von gr. *aná* („oben, aufwärts [Richtung]“, „auf etw.“) und *glyf-í* („schnitzen“) [1] – sind gegensätzlich monochrom gezeichnete und horizontal versetzte Bilder, die bei der Betrachtung mit einer geeigneten '3D-Brille' die Illusion von Räumlichkeit erzeugen.

[2] Im Allgemeinen ist die Parallaxe eines Objekts definiert als “apparent angular displacement of the object caused by viewing it from a different angle”. [19]

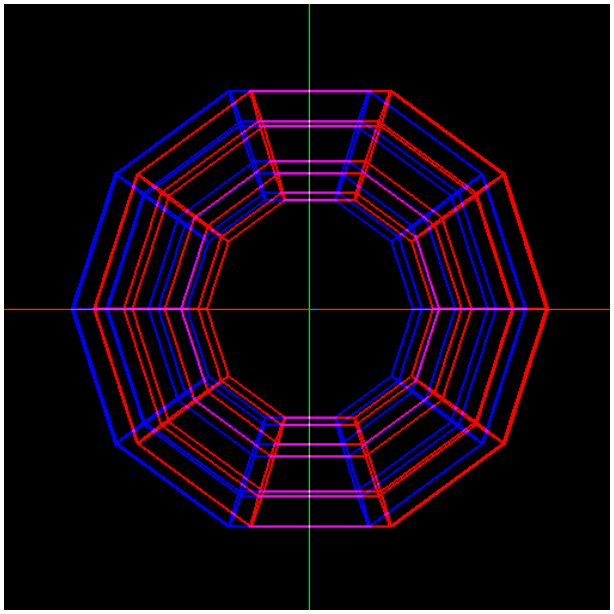


Abbildung 1.7: Versetzte Darstellung durch negative Parallaxe [4]

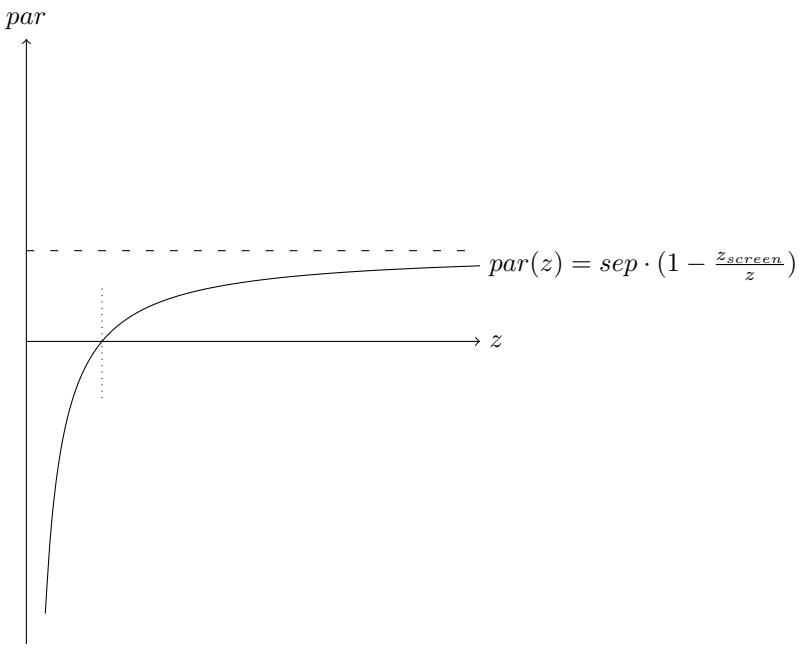


Abbildung 1.8: Die Parallaxe-Funktion [2]

Abbildung 1.9: Ein Anaglyphen-Bild (links: rot, rechts: blau, die Grünkomponente ist der Mittelwert der Grünanteile beider Seiten) [7]

Die Anaglyphen-Brille hat nach Konvention links ein rotes und rechts ein cyanfarbenes Glas[3], was dazu führt, dass bei der Konstruktion einer Anaglyphen-Anwendung auch diese Konstellation berücksichtigt werden muss. Wie in 1.2.5 beschrieben, ist es nicht egal, welche Abbildung links oder rechts auf dem virtuellen Bildschirm steht, sondern entscheidet über die Wahrnehmung von Nähe und Entfernung.

Im Folgenden soll erklärt werden, wie eine Anaglyphenbrille funktioniert und wie aus einem Anaglyphenbild die Illusion der Räumlichkeit entsteht.

1.3.1 Rot- und Blaufilter

Die Lichtstahlen aus der Umgebung (Sonne, Schreibtischlampe) treffen ständig auf Objekte in unserer Umgebung. Diese Objekte reflektieren die Lichtstrahlen gemäß ihrer Eigenfarbe (ein roter Gegenstand reflektiert fast alle Rotanteile des Lichts, dafür absorbiert er die Grün- und Blauanteile fast vollständig[4]), und diese Reflexionen erreichen dann unsere Netzhaut, wo schließlich die Farbinformation entschlüsselt wird.

Die Anaglyphenbrille filtert mit dem linken, roten Glas alle Grün- und Blauanteile der eingehenden Lichtstrahlen heraus und mit dem rechten, cyanfarbenen alle Rotanteile. Auch wenn oft der Einfachheit halber die Brille rot-blau genannt wird, ist es doch meist nötig, dass die beiden Gläser komplementär zueinander sind. Denn so entsteht auch bei Anaglyphen von Bildern mit Grünanteil der Eindruck der räumlichen Tiefe.

1.3.2 Die menschliche Farbwahrnehmung

Oben wurde schon angedeutet, dass die menschliche Netzhaut bei einem eingehenden Lichtsignal nur zwischen Rot-, Grün- oder Blauanteilen unterscheidet. Der Grund dafür sind die drei verschiedenen Arten von *coni retinae* in

[3] Es gibt auch andere Varianten wie rot-blaue, rot-grüne oder grün-magentafarbene Anaglyphen.
[4] mehr dazu in 4.1.3

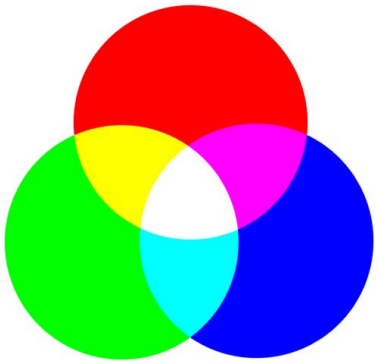


Abbildung 1.10: Das RGB-Farbmodell

der Netzhaut (Zapfenzellen), die auf eine der drei Farben besonders effektiv ansprechen [13]. Jede wahrgenommene Farbe ist somit eine Mischung dieser drei Farbkomponenten in ihrer jeweiligen Intensität. Daher erklärt sich das verbreitete RGB-Farbschema, das auch in OpenGL Verwendung findet.

Wenn nun vor dem Auge ein roter Absorptionsfilter liegt, der folglich nur rote Lichtstrahlen durchlässt, wird

- die wahrgenommene Umgebung dunkler, da die grünen und blauen Lichtanteile eliminiert werden.
- ein weißer Gegenstand als rot wahrgenommen, da die grünen und blauen Anteile des Weiß eliminiert werden und nur noch Rot übrigbleibt.
- ein blauer Gegenstand als schwarz wahrgenommen, da wieder die blauen Anteile herausgefiltert werden.
- ein roter Gegenstand als reines Rot wahrgenommen, da alle eventuellen Blau- oder Grünkomponenten verloren gehen.

Die Anaglyphenbrillen bestehen aus roten und blauen Absorptionsfiltern, die dem jeweiligen Auge eine modifizierte Variante der Szene zuführen.

1.3.3 Überlagerung und Eliminierung

Nach dem Aufsetzen der Anaglyphenbrille empfängt das linke Auge nur rote und das rechte nur blaue und grüne Farbsignale, also wird bei Betrachtung eines rot-blauen Objekts die jeweilige andere Bildhälfte eliminiert. Nun hat also das linke Auge die rote, gemäß der Parallaxe verschobene Version des Bildes erhalten, und das rechte Auge nimmt wiederum nur die blaue, entgegengesetzt verschobene Bildhälfte wahr.

Die optischen Reize, die von linker und rechter Netzhaut empfangen werden, werden schließlich stereoptisch übereinandergelagert. Dabei transportiert das linke Auge die Rotkomponente und das rechte die Blau- und Grünkomponente, woraus dann wieder ein farblich korrektes, aber dreidimensional erscheinendes Objekt ensteht.

Abbildung 1.11: Ein stereoskopisch dargestelltes Objekt, zur Betrachtung mit einer Anaglyphenbrille [4]

$$\begin{aligned} R_{final} &= R_{left} \\ G_{final} &= G_{right} \\ B_{final} &= B_{right} \end{aligned} \tag{1.1}$$

[7]

1.4 Technische Umsetzung

Nachdem die Konzepte und Grundbegriffe der Stereoskopie und Anaglyphen erläutert wurden, kann nun beides zur Konstruktion einer grafischen Umgebung genutzt werden. Dieser Abschnitt beschäftigt sich mit der Umsetzung der binokularen Sicht durch simulierte Kameras für jedes Auge.

Dazu modelliert man die Perspektive jeden Auges durch ein *Frustum*: einen Pyramidenstumpf, der die Dimensionen der Projektionsfläche beschreibt (Abbildung 1.12). In OpenGL ist die Projektionsfläche die Near Plane, von der aus man den virtuellen Bildschirm betrachtet, im Vergleich zu dem Objekte nah oder fern erscheinen [2].

1.4.1 Parallele Kamerakonfiguration

Die beiden, horizontal versetzten 'Kameras' der Augen blicken parallel zur z-Achse auf eine Szene. Jedoch sind die Frustums asymmetrisch [5]; das heißt, dass z.B. das linke Auge auf der linken Seite etwas weniger 'Projektionsspielraum' ("viewing volume" [11]) hat, also aus einem kleineren Bildschirmabschnitt gewählt wird, was auf die virtuelle linke Bildschirmhälfte abgebildet wird (Abbildung 1.13).

Gerade diese Asymmetrie ermöglicht erst das Eintreten der Parallaxe und somit die Stereopsis. Denn nur

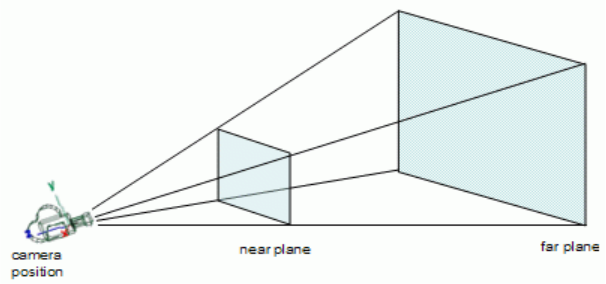


Abbildung 1.12: Frustum im dreidimensionalen Raum [22]

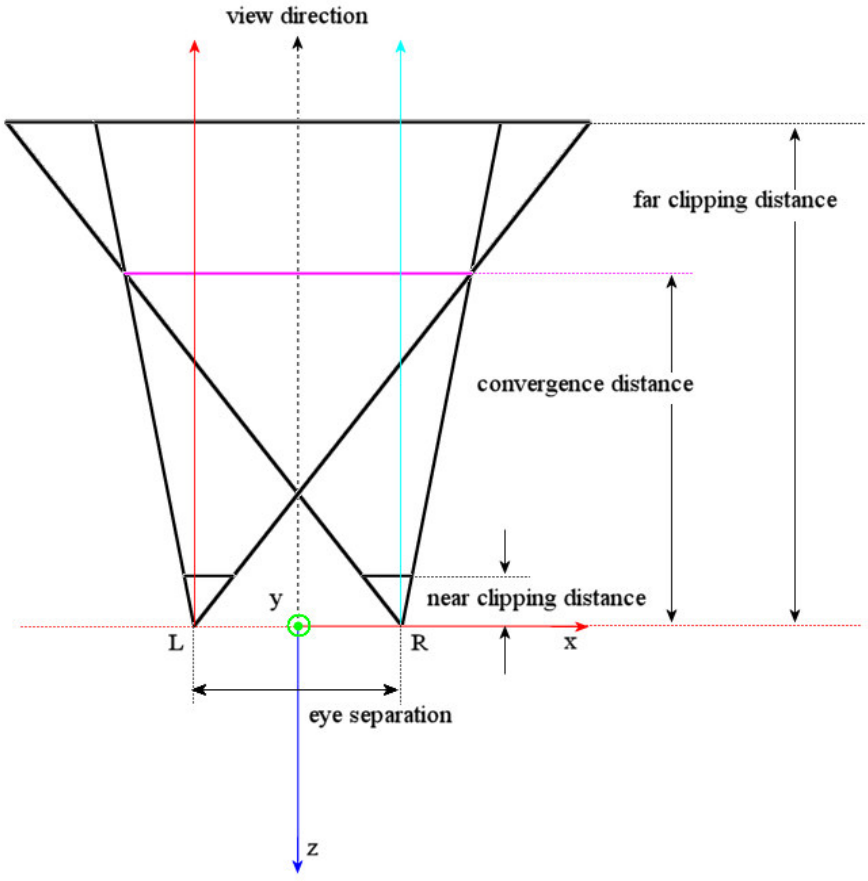


Abbildung 1.13: Lage und Ausrichtung der Frustums (Draufsicht) [5]

dadurch ist gewährleistet, dass eine Verschiebung der Abbildungen nur auf der x-Achse stattfindet. Bei geneigten Sichtachsen der Kameras („toed-in cameras“) wäre das nicht ohne weiteres gegeben:

> “Convergence by horizontal shift of the images obtained from parallel cameras introduces no distortion of horizontal or vertical screen disparity (parallax).
>
> ...
>
> It is well known that the toed-in configuration distorts the images in the two cameras producing patterns of horizontal and vertical screen disparities (parallax). Unless a pair of projectors with matched convergence or a single projector and special distortion correction techniques are used, then the projected images will have disparity distortion.” [6]

Diese “disparity distortion” ist eben der unangenehme Effekt, den man durch das Verwenden paralleler Kameras mit asymmetrischen Frustums vermeidet.

1.4.2 Strahlensatz

Im Folgenden wird gezeigt, wie die Projektion der Bildschirmebene auf die Near Plane durchgeführt wird. Dazu wird etwa in OpenGL die Entfernung, in der der virtuelle Bildschirm liegen soll, und die Entfernung zur Near Clipping Plane angegeben. Dazu benötigt man noch – für beide 'Augen' – die linke und rechte Kante der Projektionsfläche sowie die obere und untere Kante.

Wie man sehen wird, kann man die vertikalen Kanten einfach mittels eines gegeben Seitenverhältnisses (wie das verbreitete Format 16:9) aus den horizontalen Koordinaten herleiten. Diese wiederum können durch Anwendung des Zweiten Strahlensatzes (“intercept theorem”) gewonnen werden.

Wiederholung: Die beiden Augenpunkte liegen um x_{inter} horizontal voneinander entfernt, die Separation ist $sep = x_{inter}/w$. Die Sichtkegel konvergieren bei z_{screen}. Die Breite des so entstandenen virtuellen Bildschirms ist $|S_L S_R| =: w$.

Durch Betrachtung der Skizze 1.14 erkennt man, dass die Sichtgeraden des linken Auges die Near Plane und den Screen schneiden. Das ermöglicht die Anwendung des *Zweiten Strahlensatzes*:

$$\begin{aligned} \frac{l_1}{\frac{w}{2} - \frac{sep}{2}} = \frac{l_2}{\frac{w}{2} + \frac{sep}{2}} &= \frac{d_{near}}{d_{screen}} \\ l_1 &= (\frac{w}{2} - \frac{sep}{2}) \cdot \frac{d_{near}}{d_{screen}} \\ l_2 &= (\frac{w}{2} + \frac{sep}{2}) \cdot \frac{d_{near}}{d_{screen}} \end{aligned} \tag{1.2}$$

mit $d_{near} := |z_{near}|$, $d_{screen} := |z_{screen}|$, $d_{far} := |z_{far}|$.

Wie in Abbildung 1.14 zu erkennen, schneiden die Sichtgeraden die Near Plane, von L aus gesehen, bei l_1 und l_2.[5] Für die rechte Kamera kann man analog r_1 und r_2 herausfinden.

Die Grundlage des Zweiten Strahlensatzes ist die Erkenntnis, dass

- l_1 im selben Verhältnis zu $\frac{w}{2} - \frac{sep}{2}$ steht wie
- l_2 zu $\frac{w}{2} + \frac{sep}{2}$ und
- d_{near} zu d_{screen}.

[5] Vom Mittelpunkt aus gesehen, sind das die späteren `left`- und `right`-Argumente an `glFrustum()`.

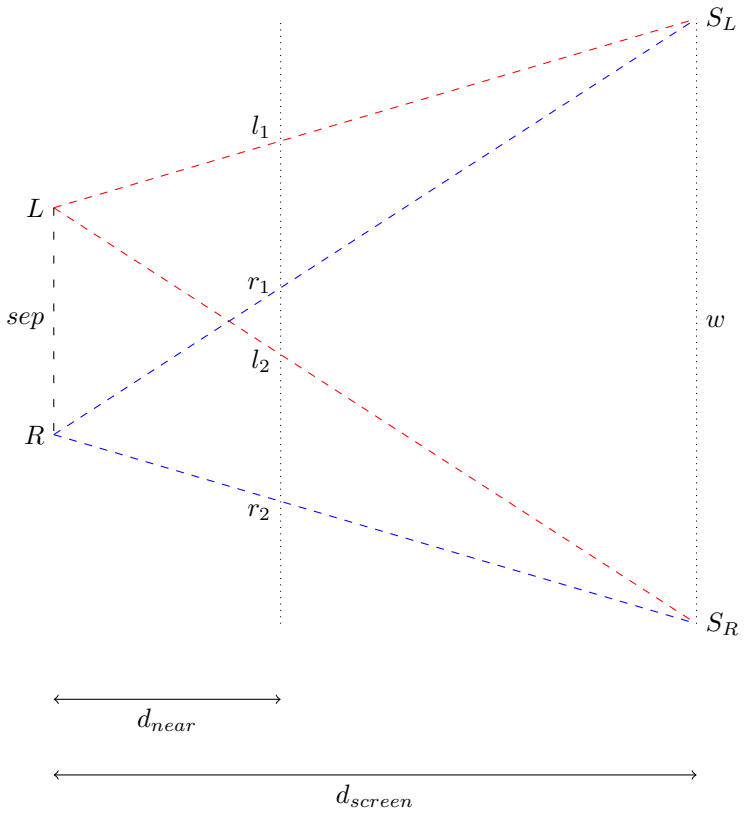


Abbildung 1.14: Skizze zur Anwendung des Zweiten Strahlensatzes für die Sichtgeraden der beiden Augen

Mit gegebenen w (s.u.), sep, z_{near} und z_{screen} ist es einfach, die Beziehung umzuformen und die Schnittpunkte der Sichtgeraden mit der Near Plane festzustellen. Daraufhin kann man mithilfe des Seitenverhältnisses $\frac{w}{h}$ auch die obere und, symmetrisch dazu, die untere Kante der Near Plane errechnen. Für das verbreitete Seitenverhältnis 16:9 geht das folgendermaßen:

$$\begin{aligned} h &= w \cdot \frac{h}{w} \\ \pm y_{near} &= \pm \frac{h}{2} \\ \text{mit } \frac{w}{h} &= \frac{16}{9} \end{aligned} \tag{1.3}$$

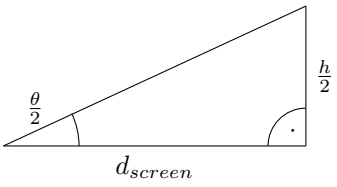
Das Seitenverhältnis kann man als Konstante definieren oder, um flexiblere Anwendungen zu erstellen, aus dem aktuellen Seitenverhältnis des Grafikfensters übernehmen.

1.4.3 Vertikaler Bildwinkel

Der vertikale Bildwinkel θ beschreibt den Winkel zwischen oberer und unterer Kante des Sichtfeldes.[6] Ein Winkel θ von z.B. $150°$ gleicht einem Fischaugenobjektiv einer Kamera.

Aus einem gegebenen θ und d_{screen} kann man die Breite des virtuellen Bildschirms w berechnen. Dazu stellt man folgende Rechnung auf:

$$\begin{aligned} \tan(\frac{\theta}{2}) &= \frac{\frac{h}{2}}{d_{screen}} \\ h &= 2 \cdot \tan(\frac{\theta}{2}) \cdot d_{screen} \\ w &= h \cdot \frac{w}{h} \\ \text{mit } \theta = 60°,\ d_{screen} &= |z_{screen}|,\ \frac{w}{h} = \frac{16}{9} \end{aligned} \tag{1.4}$$



Der Tangens ist definiert als der Quotient von Gegenkathete und Ankathete im rechtwinkligen Dreieick. Der Tangens von $\frac{\theta}{2}$ ist der Quotient aus der halben virtuellen Bildschirmhöhe und dem Abstand zum virtuellen Bildschirm. Daraus erhält man durch Umstellen und Multiplikation mit 2 die Konstante h. Mit einem gegebenen Seitenverhältnis erhält man dann die Bildschirmbreite w.

Dieses erhaltene w kann dann im Strahlensatz verwendet werden.

[6] In englischen Quellen wird für den (vertikalen) Bildwinkel die Bezeichnung "(vertical) field of view" verwendet.

Kapitel 2

Spezielle Konzepte der Computergrafik

2.1 Bézierkurven

Im Demoprogramm gibt es die Möglichkeit, eine Bézierfläche darzustellen. Diese besteht aus zwei Bézierkurven, die zusammengelegt wurden. Allgemein werden Bézierkurven wegen ihres sanften, kontrollierten Verlaufs verwendet.

Eine Bézierkurve mit $n+1$ Kontrollpunkten $p_{0..n}$ ist definiert als: [32]

$$C(u) = \sum_{i=0}^{n} \frac{n!}{i! \cdot (n-i)!} \cdot u^i \cdot (1-u)^{n-i} \cdot p_i,\ 0 \leq u \leq 1$$

Eine Bézierkurve dritter Ordnung (Grad $n = 2$) ist also:

$$C(u) = (1-u)^2 \cdot p_0 + 2 \cdot u \cdot (1-u) \cdot p_1 + u^2 \cdot p_2$$

Eine Bézierfläche wird aus zwei Bézierkurven durch das Tensorprodukt zusammengesetzt. Allgemein wird eine Bézierfläche so konstruiert: [32]

$$S(u,v) = \sum_{i=0}^{n_i} \sum_{j=0}^{n_j} \frac{n_i!}{i! \cdot (n_i - i)!} \cdot u^i \cdot (1-u)^{n_i - i} \cdot \frac{n_j!}{j! \cdot (n_j - j)!} \cdot v^j \cdot (1-v)^{n_j - j} \cdot p_{ij}$$

Bei zwei Bézierkurven dritter Ordnung (Grad $n_i = n_j = 2$) erhält man als Flächenfunktion:

$$\begin{aligned} S(u,v) = {} & u^2v^2p_{00} + 2u^2v(1-v)p_{01} + u^2(1-v)^2p_{02} \\ & + 2u(1-u)v^2p_{10} + 4u(1-u)v(1-v)p_{11} + 2u(1-u)(1-v)^2p_{12} \\ & + (1-u)^2v^2p_{20} + 2(1-u)^2v(1-v)p_{21} + (1-u)^2(1-v)^2p_{22} \end{aligned}$$

Im Allgemeinen wählt man für die Kontrollpunkte zwei- oder dreidimensionale Vektoren, die etwa in grafische Punkte umgesetzt werden können (Abbildung 2.1).

2.2 Kollision zweier Körper

Die Kollision zweier Körper bestimmt, in welchem Maß die beiden Impulse verringert oder erhöht werden, sowie die Effizienz der Kollision.

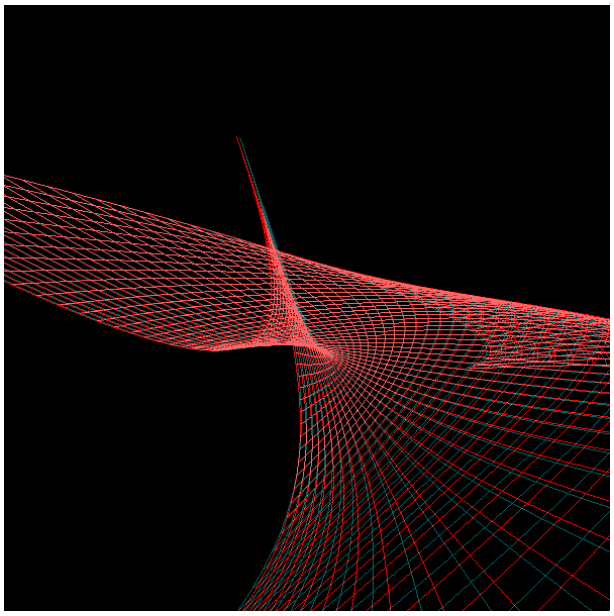


Abbildung 2.1: Im Demoprogramm realisierte Bézierfläche

Zunächst definiert man den Restitutionskoeffizienten als

$$e = \frac{v_2' - v_1'}{v_1 - v_2}$$

Er gibt an, in welchem Verhältnis die finale Geschwindigkeit zweier kollidierender Körper zur initialen Geschwindigkeit steht.

Bei einem Aufprall von Körper A auf Körper B: [21]

- Falls $e = 0$, handelt es sich um eine inelastische Kollision, bei der die kollidierenden Körper nach der Kollision zusammenhängen. A gibt die Hälfte seines Impulses an B ab und daraufhin bewegen sich beide mit der gleichen Geschwindigkeit.
- Falls $0 < e < 1$, geht kinetische Energie während des Aufpralls 'verloren'. Die Geschwindigkeit von A geht nur zu einem Teil auf B über, der Rest des Impulses bleibt in A erhalten.
- Falls $e = 1$, handelt es sich um eine elastische Kollision. Die kinetische Energie ändert sich nicht, da A seinen Impuls vollständig auf B überträgt.

Der Fall $0 < e < 1$ ist die verbreitete *teilelastische Kollision*, die auch in der präsentierten Anwendung realisiert wurde (im Programm etwa $e = 0,95$).

Bei einer erfolgten Kollision wird für die Geschwinidigkeit der beiden kollidierenden Objekte folgende Rechnung aufgestellt:

$$\begin{aligned} v_1' &= 0.5 \cdot v_1 + 0.5 \cdot v_2 + 0.5 \cdot e \cdot (v_2 - v_1) \\ v_2' &= 0.5 \cdot v_2 + 0.5 \cdot v_1 + 0.5 \cdot e \cdot (v_1 - v_2) \end{aligned} \tag{2.1}$$

2.3 Luftwiderstand

Zur Simulation von Luftwiderstand oder Reibung gibt es folgende "drag equation":

$$D = C_d \cdot \frac{\rho \cdot v^2}{2} \cdot A \quad [37] \tag{2.2}$$

Der Luftwiderstand D wirkt der Geschwindigkeit v eines Objektes entgegen. Man betrachtet hier den Betrag der Geschwindigkeit: es kann eventuell $\vec{v}$ in eine negative Richtung zeigen, aber es gilt $v = |\vec{v}|, v >= 0$.

Man kann als Vereinfachungen die Dichte ρ auf 1 setzen und den Koeffizienten C_d und die Fläche A zu einem einzigen Parameter zusammenfassen. Dadurch erhält man die folgende einfache Gleichung zur Bestimmung der neuen Geschwindigkeit:

$$v' = v - C \cdot \frac{v^2}{2}$$

Durch Justierung des Koeffizienten C lässt sich der effektive Luftwiderstand erhöhen oder verringern.

Kapitel 3

Die vierte Dimension

Ein weiterer Bestandteil des Demoprogramms ist ein Hyperspace-Modus, in dem vierdimensionale Modelle binokular betrachtet und rotiert werden können.

Mathematisch betrachtet ist es kein Problem, ein Objekt im vierdimensionalen Raum zu positionieren, zu rotieren und zu projizieren. Bei diesen Operationen werden stets nur Matrizen bearbeitet, die die Konfiguration des Objektes beschreiben. Komplizierter ist die Vorstellung eines vierdimensionalen Objekts im dreidimensionalen Raum, denn diese vierte Dimension ist zwangsläufig eine imaginäre. Nun ist es aber möglich, ein vierdimensionales Objekt in den dreidimensionalen Raum abzubilden, wobei schnell deutlich wird, wie „außerirdisch“ diese vierte Dimension tatsächlich ist.

3.1 Platonische Körper im vierdimensionalen Raum

Ein Punkt im vierdimensionalen Raum ist definiert als

$$\vec{P} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \in \mathbf{R}^4$$

, wobei die w-Koordinate genau wie die bekannten drei Koordinaten eine Ausrichtung im Raum beschreibt – aber in einem Raum höherer Dimension.

Genau, wie eine zweidimensionale Fläche in die dritte Dimension erweitert werden kann, kann auch jedes n-dimensionale Objekt in die nächsthöhere Dimension erweitert werden. Von der nullten Dimension aus: Ein Punkt x wird zur Strecke $s \in \mathbf{R}$, diese Strecke durch Erweitern in y-Richtung zu einer Fläche $F \in \mathbf{R}^2$, diese wiederum zu einem dreidimensionalen Objekt $G \in \mathbf{R}^3$. Zur Erweiterung in die vierte Dimension wählt man einfach wieder geeignete Punkte und fügt diese als w-Koordinaten ein.

3.1.1 Platonische Körper und 4-Polytope

Ein *platonischer Körper* (reguläres Polyeder) besteht aus einer bestimmten Anzahl von Ecken und Kanten in gleicher Konfiguration: alle Ecken haben gleich viele Nachbarn, alle Kanten haben die selbe Länge und alle Flächen sind gleich geformt. Zu den platonischen Körpern gehören der Würfel oder die gleichseitige Pyramide (Tetraeder).

In der vierten Dimension entstehen aus diesen beiden der Tesserakt und das Pentachoron. [33] Die regulären 4-Polytope (Polychora) sind die Erweiterungen der platonischen Körper in die vierte Dimension: ein Tesserakt

besteht aus acht würfelförmigen Zellen, die gleichmäßig miteinander verbunden sind. Ein Pentachoron besteht aus fünf tetraederförmigen Zellen. Die Zellen eines 4-Polytops sind analog zu den Flächen eines Polyeders. Sie sind die nächstkleinere geometrische 'Einheit', in die der Körper unterteilt ist.

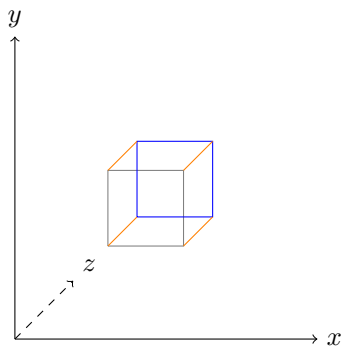
Die regulären 4-Polytope sind sehr angenehm, da sie durch ihre Gleichmäßigkeit wenige Überraschungen bei der Umsetzung in die vierte Dimension sowie bei der Betrachtung des entstandenen vierdimensionalen Objekts verursachen.

3.1.2 Konstruktion eines Tesserakts

Für den *Tesserakt* – von gr. *tessáres* („vier“) und *aktis* („Strahl“, „Speiche“) [1]; auch bekannt als Hyperwürfel, 4-Würfel oder “8-cell” – ist der Konstruktionsvorgang der folgende:

- Punkt → Strecke → Quadrat → Würfel → Tesserakt [33]

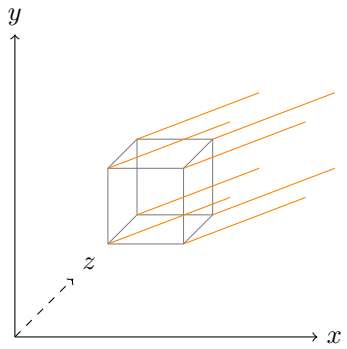
Auch wenn sich der Mechanismus einfach so aufzählen lässt, ist es dennoch nicht trivial, ihn zu verstehen. Beim Würfel angekommen, sieht unser Koordinatensystem so aus:



Im vorherigen Schritt (Gerade → Würfel) wurde gerade die graue Fläche in z-Richtung erweitert, wobei die blaue Fläche entstanden ist und zwischen den Ecken der beiden Flächen vier Verbindungskanten entstanden sind. Diese etwas merkwürdige Beschreibung umfasst alle Bedingungen, die auch für die Transformation Würfel → Tesserakt gelten müssen:

- Das Koordinatensystem wird um die w-Komponente erweitert
- Von jeder Ecke des Würfels aus geht eine Erweiterung in die neue Dimension aus
- Die neuen Ecken werden zu einem weiteren Würfel verbunden, also erhält man zwei „interdimensional“ verbundene Würfel

Die folgende Grafik zeigt das Ergebnis so deutlich wie möglich:



Die orangenen Verbindungslinien reichen in die vierte Dimension, wo aus den entstandenen Punkten der zweite Würfel zusammengefügt wird. Diese w-Koordinate lässt sich natürlich nicht in einem kartesischen Koordinatensystem darstellen, deshalb wird der Dimensionsübergang hier nur angedeutet.

Der Tesserakt hat 16 Ecken, 32 Kanten, 24 Flächen und 8 würfelförmige Zellen.

3.1.3 Konstruktion eines Pentachorons

Das *Pentachoron* – von gr. *pentás* („fünf“) und *khõros* („Ort“, „Ausdehnung“) [1]; auch bekannt als Simplex, 4-Tetraeder oder “5-cell” – ist die vierdimensionale Erweiterung eines Tetraeders. Auch hierzu gibt es eine Konstruktionsvorschrift:

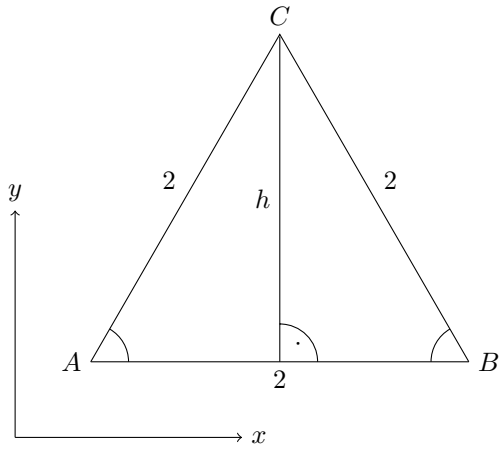
- Punkt → Strecke → Gleichseitiges Dreieck → Tetraeder → Pentachoron [33]

Man sucht bei jedem Schritt die Mitte des aktuellen Objekts und erweitert diesen Punkt so weit in die nächste Dimension, dass er gleich weit von allen Punkten entfernt ist. Als zusätzliche Herausforderung muss man nun die Mitte des Tetraeders finden und diese in die vierte Dimension erweitern.

Da diese Konstruktion schwieriger ist, lohnt es sich, sie von Schritt 2 aus nachzuverfolgen.

Von der Strecke zum gleichseitigen Dreieck

Beim Erweitern einer Strecke AB der Länge 2 nach 'oben' entsteht ein gleichseitiges Dreieck ABC der Seitenlänge 2.



Die Höhe dieses gleichseitigen Dreiecks erhält man folgendermaßen:

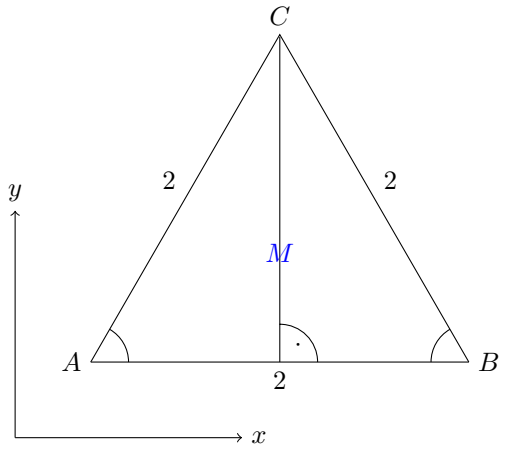
$$h = 2 \cdot \sin(60°) = 2 \sin \frac{\pi}{3} = 1,732$$

Der Punkt C ergibt sich daraus: $C(1; 1,732) \in \mathbf{R}^2$.

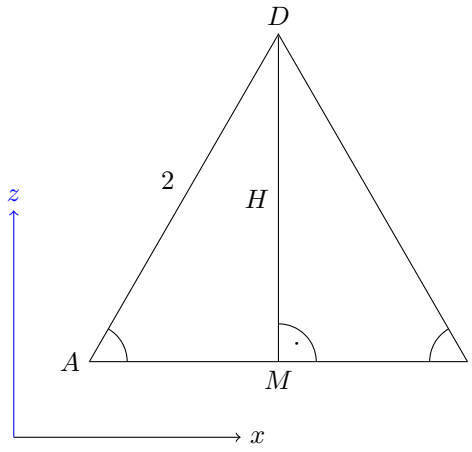
Vom gleichseitigen Dreieck zum Tetraeder

Ein gleichseitiges Dreieck der Seitenlänge 2 hat seinen Mittelpunkt bei:

$$M(\frac{A_x + B_x + C_x}{3}; \frac{A_y + B_y + C_y}{3}) = M(\frac{3}{3}; \frac{1,732}{3}) = M(1; 0,577) \in \mathbf{R}^2$$



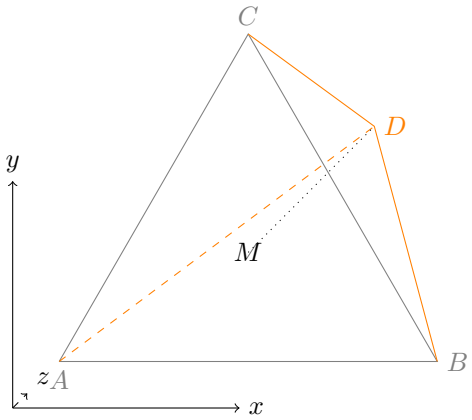
Dieser Mittelpunkt wird nun in z-Richtung erweitert. Die z-Koordinate des neuen Punktes D ergibt sich wieder aus einem gleichseitigen Dreieck – diesmal eines, das das entstandene Tetraeder im dreidimensionalen Raum durchquert.



Man kann wieder die bekannte Gleichung verwenden, um die Höhe des Tetraeders und daraus den Punkt D zu erhalten:

$$H = 2 \sin \frac{\pi}{3} = 1,732 \Rightarrow D(1; 0,577; 1,732) \in \mathbf{R}^3$$

Das entstandene Tetraeder sieht so aus:



Vom Tetraeder zum Pentachoron

Nun ist der Mittelpunkt des Tetraeders zu bestimmen. Von diesem aus soll das Tetraeder dann in w-Richtung erweitert werden. Analog zum Mittelpunkt M eines gleichseitigen Dreiecks kann man auch den Mittelpunkt μ eines Tetraeders erhalten:

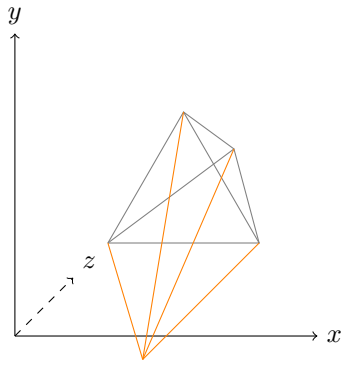
$$\mu(\frac{A_x + B_x + C_x + D_x}{4}; \frac{A_y + B_y + C_y + D_y}{4}; \frac{A_z + B_z + C_z + D_z}{4})$$
$$= \mu(1; \frac{1,732 + 0,577}{2}; \frac{1,732}{4}) = \mu(1; 0,577; 0,433)$$
$$\text{mit } A(0;0;0), B(2;0;0), C(1;1,732;0), D(1;0,577;1,732)$$

Bis jetzt wurde der Mittelpunkt einer Form immer um $h = H = 1,732$ in die neue Dimension erweitert. Da es sich bei einem Tetraeder nur um eine Zusammensetzung von gleichseitigen Dreiecken handelt, kann man annehmen, dass auch die Höhe des Pentachorons

$$\eta = H = h = 1,732$$

ist. Dadurch ergibt sich als neuer Punkt $E(1;0,577;0,433;1,732) \in \mathbf{R}^4$.

Bei dieser Konstruktion hat man als Zwischenergebnis immer einen Punkt erhalten: Im Gegensatz zum Tesserakt entsteht in jeder neuen Dimension nur ein einzelner Punkt, der durch Erweiterungskanten mit den alten Punkten verbunden bleibt. Nach diesem Schema kann man nun wieder das Pentachoron andeuten:



Die orangenen Kanten reichen wieder in die vierte Dimension, wo sie im Punkt E zusammentreffen. Dieses 4-Polytop ist einfacher zu implementieren als der Tesserakt, aber dafür ist die Herleitung der Koordinaten hier deutlich aufwendiger. Aber die Umsetzung in noch höhere Dimensionen ist beim Pentachoron jetzt leicht möglich: man berechnet wieder den Mittelpunkt des Pentachorons und erweitert es um einen Punkt F in t-Richtung.

Das Pentachoron hat 5 Ecken, 10 Kanten, 10 Flächen und 5 tetraederförmige Zellen.

3.1.4 Rotation im vierdimensionalen Raum

Die Rotation eines Punktes im n-dimensionalen Raum ist die Multiplikation einer $n \times n$-Rotationsmatrix mit dem Vektor des Punktes. OpenGL führt so etwa mit der `glRotate*`-Anweisung eine Rotation im dreidimensionalen Raum durch. Allgemein sagt man häufig, dass ein Objekt um eine Achse rotiert wird – und im dreidimensionalen Raum ist diese Aussage auch unproblematisch. Aber im vierdimensionalen Raum reicht diese Beschreibung nicht aus: durch den zusätzlichen Freiheitsgrad w gibt es nicht mehr nur drei Bewegungsrichtungen, sondern sechs. [34]

Deshalb wird im Folgenden eine Rotation nicht als Rotation um eine Achse, sondern als Rotation parallel zu einer Ebene definiert. So gibt es im vierdimensionalen Raum sechs Drehrichtungen: entlang der xy-, yz-, xz-, xw-, yw- und zw-Ebene.

Für jede dieser Rotationen gibt es eine eigene Rotationsmatrix. Bei einer Rotation um den Winkel a:

$$
\begin{aligned}
xy &: \begin{pmatrix} \cos a & \sin a & 0 & 0 \\ -\sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad
yz : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & \sin a & 0 \\ 0 & -\sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad
xz : \begin{pmatrix} \cos a & 0 & -\sin a & 0 \\ 0 & 1 & 0 & 0 \\ \sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
xw &: \begin{pmatrix} \cos a & 0 & 0 & \sin a \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin a & 0 & 0 & \cos a \end{pmatrix} \quad
yw : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & 0 & -\sin a \\ 0 & 0 & 1 & 0 \\ 0 & \sin a & 0 & \cos a \end{pmatrix} \quad
zw : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos a & -\sin a \\ 0 & 0 & \sin a & \cos a \end{pmatrix}
\end{aligned}
\tag{3.1}
$$

Die xy-, yz- und xz-Rotationsmatrizen sind schon aus dem $\mathbf{R}^3$ bekannt, wo einfach die zu w gehörige Reihe und Spalte wegfällt. Die neuen drei Matrizen betreffen folglich die neue Bewegungsrichtung. Wenn also ein Punkt in eine der sechs möglichen Richtungen rotiert werden soll, stellt man folgende Gleichung auf:

$$\vec{P'} = R \cdot \vec{P}$$

Zum Beispiel:

$$
\begin{aligned}
\vec{P} &= \begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \end{pmatrix}, \text{ Rotation um } yw\text{-Ebene, } a = \frac{\pi}{3} \\
\vec{P'} = R_{yw} \cdot \vec{P} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & 0 & -\sin a \\ 0 & 0 & 1 & 0 \\ 0 & \sin a & 0 & \cos a \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 \\ -\cos\frac{\pi}{3} - \sin\frac{\pi}{3} \\ 1 \\ -\sin\frac{\pi}{3} + \cos\frac{\pi}{3} \end{pmatrix} = \begin{pmatrix} 1 \\ -1,366 \\ 1 \\ -0,366 \end{pmatrix}
\end{aligned}
\tag{3.2}
$$

Ein Vorzeichenwechsel der w-Koordinate kann dazu führen, dass ein Punkt aus dem sichtbaren Bereich hinaus rotiert wird, was das ungewöhnliche Verhalten eines rotierenden Tesserakts erklärt.

3.2 Grafische Darstellung eines vierdimensionalen Objekts

Wir haben festgestellt, dass die mathematischen Bedingungen für ein vierdimensionales Objekt nicht sehr kompliziert sind. Bei der Betrachtung einer dreidimensionalen Abbildung eines solchen Objekts ist man jedoch intuitiv nicht fähig, das Dargestellte zu begreifen.

So wie ein zweidimensionales Wesen bei der Rotation eines Würfels vor ein Rätsel gestellt ist, kann auch eine 3D-Anwendung kein 4D-Objekt natürlich darstellen. Tatsächlich sind beide Fälle ähnlich: Bei der Rotation eines Würfels scheint es für das zweidimensionale Wesen, als würde sich ein Quadrat umstülpen, während für den dreidimensionalen Betrachter klar ist, dass sich der Würfel einfach entlang der xz- oder yz-Ebene dreht. Aber dieser dreidimensionale Betrachter kann sich nicht die Rotation eines 4-Polytops erklären – oder wenigstens nicht angemessen beobachten. [35]

3.2.1 Projektion in die dritte Dimension

So wie man ein Polyeder auf eine Ebene projizieren kann, um eine flache Abbildung dessen zu erhalten – etwa die Abbildung auf einen Computerbildschirm –, kann man ein 4-Polytop in das kartesische Koordinatensystem projizieren. Beide Vorgänge sind einander sehr ähnlich; man verwendet entweder die Parallel- oder die Perspektivprojektion, um die Punkte in eine niedrigere Dimension zu versetzen.

- Parallelprojektion – Die Parallelprojektion nimmt keine Rücksicht auf eventuelle Verkürzungen der Kanten aufgrund der Perspektive. Überzählige Komponenten des Vektors $\vec{P}$ werden verworfen. [34]

$$P'_x = P_x,\ P'_y = P_y,\ P'_z = P_z; \quad P \in \mathbf{R}^4,\ P' \in \mathbf{R}^3$$

- Perspektivische Projektion – Die perspektivische Projektion passt die Kantenlängen an, um ein Gefühl der Perspektive zu erzeugen. Die überzähligen Komponenten von $\vec{P}$ dienen als Divisor für die zu erhaltenden Komponenten. [34]

$$P'_x = \frac{P_x}{P_w},\ P'_y = \frac{P_y}{P_w},\ P'_z = \frac{P_z}{P_w}; \quad P \in \mathbf{R}^4,\ P' \in \mathbf{R}^3$$

OpenGL wendet bei Aufrufen an `glVertex4*` die Formel zur perspektivischen Projektion automatisch an. Dazu werden die *homogenen Koordinaten* eingesetzt. [36]

Homogene Koordinaten in OpenGL Jeder Punkt $\vec{p}$ hat in OpenGL vier Koordinaten: (x, y, z, w), wobei bei dreidimensionalen Transformationen oder Rotationen immer $w = 1$ gilt. So werden die x-, y-, und z-Koordinaten nicht durch w beeinflusst.
Dieses System kann man sehr gut verwenden, um eine Abbildung $\mathbf{R}^4 \to \mathbf{R}^3$ vorzunehmen:

$$\vec{p'} = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}); \quad \vec{p'} \in \mathbf{R}^3,\ w \neq 0$$

Wenn w kleiner wird, werden die drei anderen Koordinaten größer, und umgekehrt. Wenn $w = 0$ gilt, wird der Punkt in der grafischen Anwendung an eine unendlich weit entfernte Stelle verschoben. Genau so verhält sich natürlich auch die oben gezeigte perspektivische Projektion. Dieses Verhalten veranschaulicht sehr gut die Unmöglichkeit eines vierdimensionalen Objekts in einer dreidimensionalen Umgebung: die aus der vierten Dimension kommenden Punkte kommen aus der 'Unendlichkeit'.

3.2.2 Behandeln von Dimensionsübergängen

- Kanten, bei denen für beide Ecken $w < 0$ gilt, sollten in der grafischen Anwendung nicht gezeigt werden.

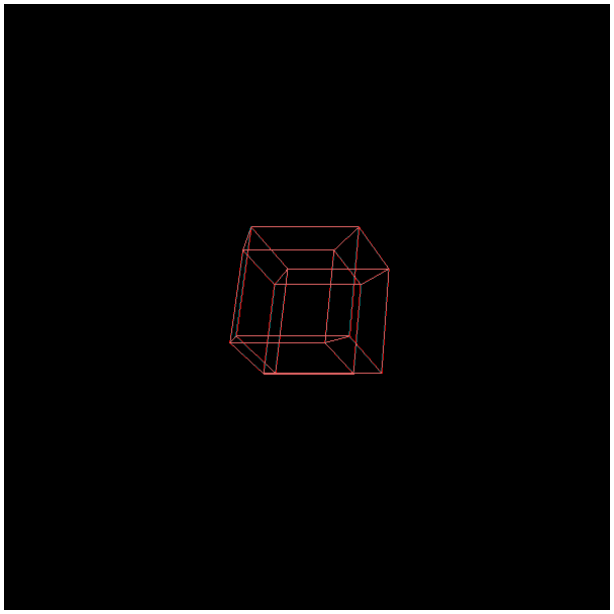


Abbildung 3.1: Tesserakt in Parallelprojektion

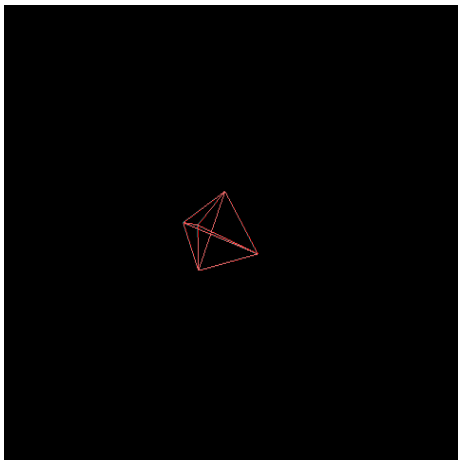


Abbildung 3.2: Pentachoron in Parallelprojektion

- Kanten mit beiden $w > 0$ können normal angezeigt werden.
- Kanten, bei denen nur eine Ecke $w < 0$ hat, müssen auf die $w = 0$-Hyperebene projiziert werden. [34]

Der dritte Fall ist etwas komplizierter zu behandeln als die ersten beiden: dazu konstruiert man eine Projektionsgerade von $\vec{P}_1$ nach $\vec{P}_2$, wobei der Schnittpunkt dieser Projektionsgeraden mit der $w = 0$-Hyperebene das Ergebnis der Projektion ist.

Die Projektionsgerade

Man stellt eine Projektionsgerade auf mit

$$g : \vec{x} = (1 - \mu)\vec{P}_1 + \mu\vec{P}_2; \quad g \in \mathbf{R}^4 \tag{3.3}$$

Für $\mu = 0$ ist man am Ausgangspunkt der Projektion, für $\mu = 1$ ist man am zu projizierenden Punkt. In der Anwendung ist es sinnvoll, die Ecke mit positiver w-Komponente als Ausgangspunkt $\vec{P}_1$ zu nehmen und die Projektionsgerade zur anderen Ecke aufzuspannen.

Die $w = 0$-Hyperebene

Diese Hyperebene ist durch folgende Gleichungen beschrieben:

$$\begin{aligned} E : \vec{x} &= \begin{pmatrix}0\\0\\0\\0\end{pmatrix} + r\begin{pmatrix}1\\0\\0\\0\end{pmatrix} + s\begin{pmatrix}0\\1\\0\\0\end{pmatrix} + t\begin{pmatrix}0\\0\\1\\0\end{pmatrix} \\ E : (\vec{x} - &\begin{pmatrix}0\\0\\0\\0\end{pmatrix}) \cdot \begin{pmatrix}0\\0\\0\\1\end{pmatrix} = 0 \Rightarrow \vec{x} \cdot \begin{pmatrix}0\\0\\0\\1\end{pmatrix} = 0 \\ E : 0x + 0y + 0z + 1w &= 0 \Rightarrow w = 0 \end{aligned} \tag{3.4}$$

Die oben gezeigten drei Ebenengleichungen sind äquivalent. Man definiert zuerst die Ebene in Parameterform, da diese intuitiv möglich ist: die Hyperebene hat als Stützvektor den Ortsvektor $\vec{0}$ und dehnt sich in x-, y- und z-Richtung aus. w ist 0, da man sich für den Übergang von positiven zu negativen w interessiert (also den Schnittpunkt der Gerade mit $w = 0$).

Als nächstes formt man die Ebenengleichung in Normalenform um. Das geht hier sehr einfach, da die Normale zu den drei Einheitsvektoren natürlich in die vierte Richtung zeigen muss. Eine weitere Vereinfachung bringt der Stützvektor $\vec{0}$, der einfach wegfällt.

Im letzten Schritt wird das innere Produkt von $\vec{x}$ und $\vec{n}$ gebildet, woraus sich die sehr einfache Ebenengleichung $E : w = 0$ ergibt. Diese in Koordinatenform vorliegende Gleichung dient nun dazu, den Schnittpunkt zu erhalten.

Schnittpunkt von Gerade und Hyperebene

Der Schnittpunkt ist dank der einfachen Ebenengleichung nicht schwer zu bestimmen. Die Rechnung gilt allgemein für die beiden Punkte $\vec{P}_1 = (x_1, y_1, z_1, w_1)$ und $\vec{P}_2 = (x_2, y_2, z_2, w_2)$.

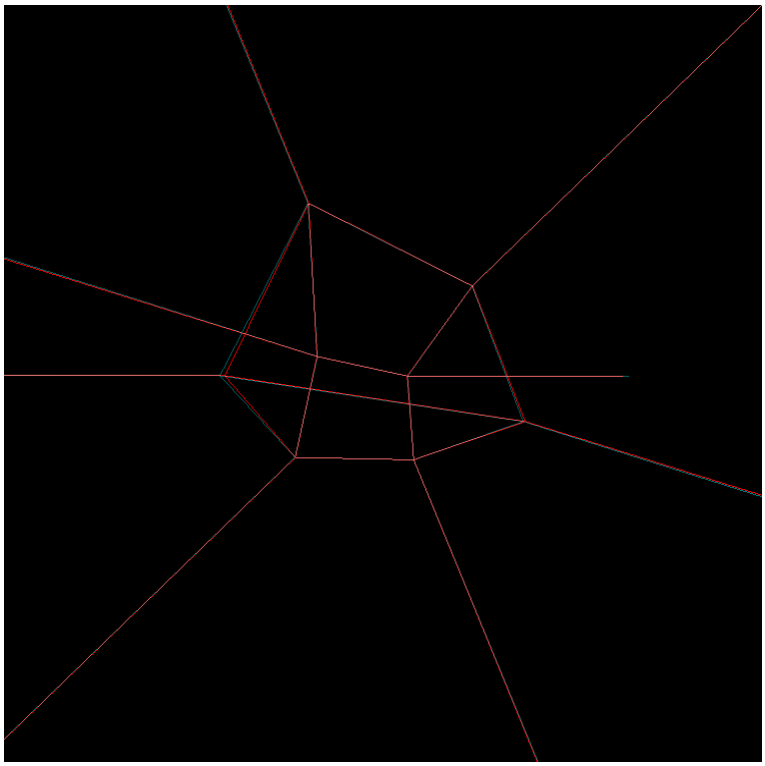


Abbildung 3.3: Tesserakt in Perspektivenprojektion, mit w-Clipping gegen $0,001$

1. Man setzt g in E ein und erhält den Faktor μ:

$$\begin{aligned} E : (1-\mu)w_1 + \mu w_2 &= 0 \\ w_1 - \mu w_1 + \mu w_2 &= 0 \\ \mu(w_2 - w_1) &= -w_1 \\ \mu &= -\frac{w_1}{w_2 - w_1} \end{aligned} \tag{3.5}$$

2. Das erhaltene μ wird wieder in g eingesetzt und man erhält den abgebildeten Punkt P':

$$g : \vec{x} = (1 - \frac{-w_1}{w_2 - w_1})\vec{P}_1 + \frac{-w_1}{w_2 - w_1}\vec{P}_2 = \vec{P'}$$

Zur Anwendung in OpenGL ergibt das Verwenden von $E : w = 0,001$ bessere Ergebnisse: Man sucht also nur 'ungefähr' den Schnittpunkt mit der $w = 0$-Hyperebene. Durch diese kleine Toleranzschwelle kann man verhindern, dass OpenGL Punkte in der Unendlichkeit generiert: dadurch bleibt die Richtung der Geraden erkennbar und man kann sich dennoch vorstellen, dass sie in sehr weite Entfernung reicht.

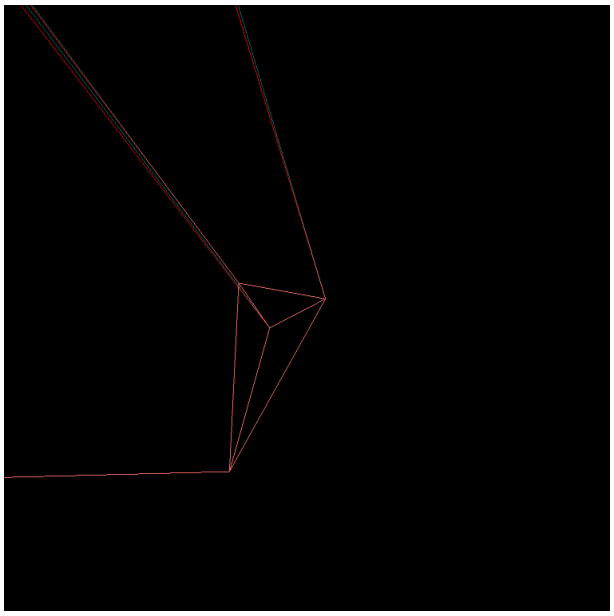


Abbildung 3.4: Pentachoron in Perspektivenprojektion, mit w-Clipping gegen $0,001$

Kapitel 4

Code Review

Nachdem in Kapitel 1 die grundlegenden Ideen der Stereoskopie erklärt wurden, soll nun die Umsetzung mittels GLUT besprochen werden. Anhand jedes relevanten Befehls wird jeweils ein Konzept von OpenGL und GLUT erläutert.

Die Betrachtung des Codes startet in der Prozedur `main` und folgt dann dem Programmfluss.

4.1 Initialisierung

4.1.1 Darstellungsmodus

Nach der Initialisierung des GLUT mit `glutInit(&argc,argv)` folgt sogleich die Initialisierung des Display Mode:

```
glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
```

- `GLUT_RGBA` bestimmt die Pixelbreite der Anwendung: hier hat jeder Pixel die vier Komponenten Rot, Grün, Blau und Alpha (Transparenz).
- `GLUT_DOUBLE` ist aktiviert. Die Anwendung verwendet Double Buffering: die berechneten Pixel werden nicht direkt auf den Bildschirm übertragen, sondern am Ende mit `glutSwapBuffers()` simultan angezeigt. Dadurch wird Tearing, die ruckelnde Darstellung der Objekte aufgrund des zu schnellen Darstellens, verhindert. Dazu dient auch die Anweisung `glDrawBuffer(GL_BACK)`, die den 'hinteren' Farbpuffer als Ziel von Zeichenoperationen auswählt. [12] Dieser wird dann beim Austauschen nach vorne geholt.
- `GLUT_DEPTH` ermöglicht das gezielte Zeichnen von Objekten im Hintergrund, die sonst andere Objekte verdecken würden. Dazu muss vor der Darstellung noch das `GL_DEPTH_BUFFER_BIT` zurückgesetzt werden.

4.1.2 Hintergrundfarbe

Etwas weiter steht die Zeile

```
glClearColor(0.0,0.0,0.0,1.0);
```

Damit setzt man die Hintergrundfarbe der Anwendung auf Schwarz. Die Rot-, Grün- und Blauanteile sind alle `0.0`, und der Hintergrund ist vollständig deckend.

4.1.3 Belichtung

Das Lichtsystem wird folgendermaßen initialisiert:

```
glEnable(GL_LIGHTING);
glLightfv(GL_LIGHT0,GL_AMBIENT,lightAmbientIntensity);
glLightfv(GL_LIGHT0,GL_DIFFUSE,lightDiffuseIntensity);
glLightfv(GL_LIGHT0,GL_SPECULAR,lightSpecularIntensity);
glLightfv(GL_LIGHT0,GL_POSITION,lightPosition);
glEnable(GL_LIGHT0);
```

Wenn nicht `GL_LIGHTING` aktiviert wird, sind alle `glLight*(...)`-Befehle wirkungslos. Um eine Farbe darzustellen, müsste man dann durch Aufruf der Prozedur `glColor*(...)` die Farbe eines grafischen Objektes verändern.

Zuerst seien hier die Aufrufe mithilfe der OpenGL-Definition[8] erklärt:

- `glLightfv` akzeptiert folgende Argumente:

`GLenum light`	die Bezeichnung der Lichtquelle, die man bearbeiten möchte. Wie üblich, kann man mit durch das bitweise ODER (`...|...`) die Konstanten aneinanderreihen.
`GLenum pname`	der Teil der Lichtkonstante, der geändert werden soll. Das sind entweder die drei Reflektionstypen (Ambient, Diffuse, Specular) oder die Position des Lichts.
`const GLfloat *params`	ein "float vector"(wie in der Funktionssignatur gefordert), der die Werte enthält, die den gegebenen Teil der gegebenen Lichtquelle ersetzen sollen.

- `glMaterialfv` ist folgendermaßen definiert:

`GLenum face`	bestimmt, welche Seiten der grafischen Objekte mit den gewünschten Eigenschaften überzogen werden sollen. [10]
`GLenum pname`	wie schon in der vorigen Prozedur, lassen sich die Teile des Lichts angeben, die von der Prozedur bearbeitet werden.
`const GLfloat *params`	wiederum ein Vektor aus Fließkommazahlen, der die Daten enthält.

Im oben gezeigten Codeausschnitt wird die Lichtquelle `GL_LIGHT0` mit verschiedenen Parametern eingerichtet. Es ist mit einiger Geduld verbunden, die Lichtverhältnisse zufriedenstellend zu konstruieren, die sich sowohl aus der Intensität als auch der Position der Lichtquelle ergibt. Daraufhin wird die Materialbeschaffenheit der dargestellten Objekte angepasst. Diese beiden Konzepte werden in Anhang A.2 näher erläutert.

Daraufhin wird die Lichtquelle durch `glEnable(GL_LIGHT0)` aktiviert.

Der Aufbau der Lichtquelle führt natürlich noch nicht zu einer stereoskopischen Darstellung. Dazu werden im Verlauf des Programms Farbmasken verwendet, die versetzt beide Farbanteile darstellen. Die genaue Prozedur der stereoskopischen Darstellung findet sich in Kapitel 4.4.6.

4.2 Definition der Event-Handler

Nach den anfänglichen Einrichtungen und der Übernahme des Programmflusses durch das GLUT-System kann man nur noch durch registrierte Callbacks mit diesem interagieren [17]. Etwa definiert man eine Prozedur `keyboard`, die nun immer aufgerufen wird, wenn GLUT einen Tastendruck registriert.

Nach der Initialisierung des Lichtsystems definiert man die vier wichtigen Callbacks:

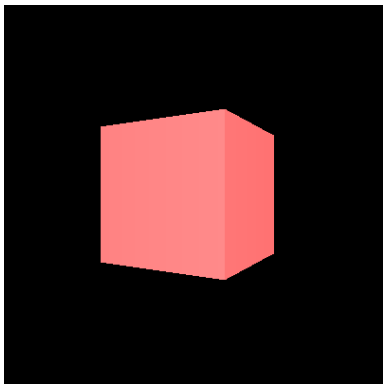


Abbildung 4.1: Beispielhafte Beleuchtung eines grafischen Objekts

```
glutDisplayFunc(display);
glutKeyboardFunc(keyboard);
glutReshapeFunc(reshape);
glutTimerFunc(5,timer,0);
```

Der Prototyp jedes Event-Handlers folgt diesem Schema: [9]

```
void glut[...]Func(void (*func)(...))
```

Die Callbacks akzeptieren verschiedene Argumente, z.B. erhält die Keyboard-Funktion den eingegebenen ASCII-Code und zwei Koordinaten:

```
void glutKeyboardFunc(
    void (*func)(unsigned char key, int x, int y));
```

4.2.1 Das Menü

Als nächstes wird das Menü eingerichtet, das bei einem Betätigen der rechten Maustaste auftaucht. Ein Auszug aus dem Quellcode zeigt die Funktionalität:

```
cubeMenu=glutCreateMenu(menu);
glutAddMenuEntry("Rotate on/off",4);
[...]
glutCreateMenu(menu);
glutAddSubMenu("Cube",cubeMenu);
[...]
glutAddMenuEntry("Stereo/Mono",5);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Es gibt die Möglichkeit, beliebig verschachtelte Submenüs zu erstellen: wie oben angedeutet, wird der Variable `cubeMenu` ein 'Identifikator' zugeordnet, der als Rückgabewert von

```
int glutCreateMenu(void (*func)(int));
```

[9] entsteht. Das Argument dieser Funktion ist das Menü-Callback `menu`. Bevor nun `cubeMenu` verwendet wird, wird zunächst mithilfe von

```
void glutAddSubMenu(const char *label,int submenu)
```

dem Submenü eine Zahl von Einträgen hinzugefügt. Natürlich könnte man weitere hierarchische Ebenen einbauen, aber das ist in dieser einfachen Anwendung nicht nötig.

Nach dem Erstellen und Aufbewahren der Submenüs in Variablen kommt man zum Wurzelmenü, das sich allein dadurch auszeichnet, dass es alle anderen Menüs beinhaltet (`glutAddSubMenu(...)`) und schließlich der rechten Maustaste zugeordnet wird.

Die entstehenden Opcodes werden von der Prozedur `menu` behandelt (in 4.6).

4.2.2 Die Main Loop

Die letzte Anweisung der Prozedur `main` ist

```
glutMainLoop();
```

Damit wird der Programmfluss endgültig von GLUT übernommen. Die Callbacks werden nach Betreteten der GLUT-Main-Loop die einzige Möglichkeit des Entwicklers sein, den Programmverlauf zu beeinflussen. [17]

4.3 Die Callbacks

Über die definierten Callbacks hat der Programmierer definiert, was die Anwendung tun soll. Die Prozeduren werden aufgerufen, sobald das GLUT-System eine Operation durchführen muss, die nicht standardmäßig definiert ist: das heißt, jegliches benutzerdefiniertes Verhalten (wie etwa das Vergrößern des Fensters oder eine Tastatureingabe).

- Display: aufgerufen, wenn das Fenster erneut gezeichnet werden muss. Entweder stellt GLUT fest, wann dieser Fall eintritt oder der Entwickler ruft die Prozedur `glutPostRedisplay()` auf, um einen Redisplay zu erzwingen.
- Keyboard: Bei einer Tastatureingabe in das GLUT-Fenster wird diesem Callback der generierte ASCII-Code übermittelt.
- Menu: durch `glutCreateMenu` erstelltes Callback, das aufgerufen wird, wenn ein Menüeintrag ausgewählt wird.
- Reshape: Wenn das Fenster verformt wird (Reshape), wird an diese Prozedur die neue Breite und Höhe übergeben. Diese Prozedur wird automatisch beim Erstellen eines neuen Fensters ausgeführt.
- Timer: Diese Funktion wird von GLUT zur spezifizierten Zeit aufgerufen, wobei auch ein Argument übergeben werden kann. Die Verwendung des Timers ist zur Animation besser geeignet als die Idle-Funktion, weil annähernd genau bekannt ist, wie oft das Callback pro Zeit aufgerufen wird.
- Idle: Callback für Hintergrundtasks und durchgehende Animationen. Diese Prozedur wird immer dann ausgeführt, wenn keine anderen Events eintreffen. Üblicherweise sollte man das Berechnen und Rendern innerhalb der Idle-Funktion auf weniger als einen Frame Rechenzeit beschränken.

[20]

4.4 Die Display-Prozedur

Die Display-Prozedur ist sicherlich das wichtigste Callback des Demoprogramms. Hier gilt es zu definieren, was in jedem Frame der Animation geschehen soll.

Hier folgt ein grober Überblick über die Tätigkeiten der Display-Prozedur:

- Leeren der Puffer
- Einrichten von durch den Benutzer gewählten Optionen (Culling, Wireframe)
- Einrichten der Projektion
- Darstellen der Szene
  - falls man im stereoskopischen Modus ist: Szene für jedes 'Auge' separat rendern
- Puffer austauschen

4.4.1 Einleitende Deklarationen

```
double znear=-0.1,zscreen=-10.0,zfar=-100.0,eye=0.0; /*zscreen: convergence*/
double w=2*dscreen*tan(fov/2)*(width/height); /* virtual screen width */
float fov=60.0*PI/180.0; /* vertical(!) field of view */
double sep=0.9/w; /* eye separation - tune to adjust parallax */
```

Analog zu Kapitel 1.4.2 werden hier die Konstanten z_{near}, z_{screen} (Konvergenz) und z_{far} deklariert. Die Variable eye wird später für die horizontale Verschiebung der Augenpunkte verwendet.

Die Breite des virtuellen Bildschirms w berechnet sich aus dem in Kapitel 1.4.3 erklärten Zusammenhang mit dem vertikalen Bildwinkel `fov`. Die C-Bibliotheksfunktion `double tan(double x)` erwartet ihr Argument im Bogenmaß, daher formt man die gewünschten $60°$ noch in Radians um.

Die Konstante sep ist x_{inter}/w (s. 1.2.3), wobei hier x_{inter} beliebig auf `0.9` festgelegt wurde, da dieser Wert einen annehmbaren stereoskopischen Effekt erzeugt (ohne zu große oder zu kleine Parallaxe).

4.4.2 Leeren der Puffer

Nach den einführenden Deklarationen steht die Anweisung:

```
glClear(GL_COLOR_BUFFER_BIT);
if(!stereo) glClear(GL_DEPTH_BUFFER_BIT);
```

So werden die von OpenGL verwendeten Puffer geleert, und zwar mit zuerst der Farbpuffer mit der Farbe, die in 4.1.2 festgelegt wurde (in diesem Fall: schwarz). Dann setzt man, falls man im monokularen Betrieb ist, den z-Puffer zurück. Falls nicht, werden später selektiv für die rote und blaue Abbildung die Tiefenwerte zurückgesetzt.

Auf diesen Hintergrund werden im Folgenden die Objekte gezeichnet.

Weitere Informationen zu den OpenGL-Puffern finden sich in A.3.

4.4.3 Sichtbarkeit und Verdeckung

Wir befinden uns immer noch in den initialen Einstellungen, die vor jedem gezeichneten Frame überprüft werden.

```
if(e.depth_test) glEnable(GL_DEPTH_TEST);
else glDisable(GL_DEPTH_TEST);
```

Falls die Option `e.depth_test` im Menü aktiviert wurde (4.6), wird nun der Depth Test (`GL_DEPTH_TEST`) aktiviert. Bei einem aktivierten Depth Test gibt es eine simulierte Verdeckung des Hintergrunds durch den Vordergrund.

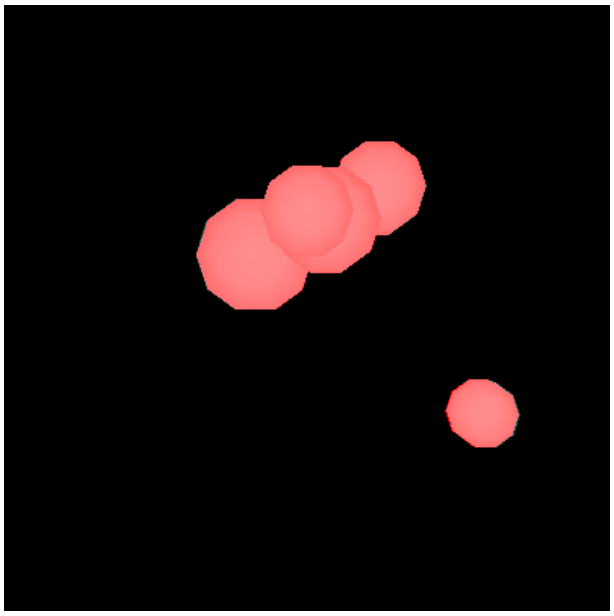


Abbildung 4.2: Deaktivierter Depth Test

Der Depth Test Nach dem Leeren des z-Puffers ist jede Speicherstelle mit einem Wert belegt, der so weit wie möglich vom Augenpunkt entfernt ist. [12] Dann wird für jedes Pixel jedes Objekts überprüft, ob dieses eine niedrigere Entfernung zum Augenpunkt als das gespeicherte hat, sodass sich am Ende nur die nächsten Pixelwerte im Tiefenspeicher befinden. Dadurch erzielt OpenGL eine *Hidden Surface Removal*, die Verdeckung von weiter hinten liegenden Objekten.

Wenn der Depth Test deaktiviert ist, hängt die Verdeckung allein von der Reihenfolge, in der die Objekte gezeichnet werden, ab: die Pixels des zuletzt berechneten Objekts verdecken im Farbpuffer alle, die schon an der jeweiligen Stelle vorhanden sind. Dadurch verdeckt das vorderste Objekt alles, das eventuell vorher an der gleichen Position gezeichnet werden sollte.

Darauf folgt dieser Code:

```
if(e.cull_face) glEnable(GL_CULL_FACE);
else glDisable(GL_CULL_FACE);
```

Das Back Face Culling, also das Vernichten von hinten liegenden Flächen, dient etwa zum Darstellen von ausgeschnittenen geometrischen Objekten, z.B. bei einem Seitenriss eines Hauses.

4.4.4 Der Polygonmodus

Der Polygonmodus bestimmt, ob für Vorder- und Rückseite die gleiche Darstellungsweise verwendet wird. Die Prozedur

`void glPolygonMode(GLenum face, GLenum mode);` [8]

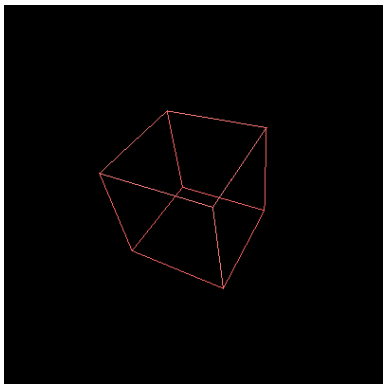


Abbildung 4.3: Deaktiviertes Back Face Culling

akzeptiert als `face` die Konstanten `GL_FRONT`, `GL_BACK` oder `GL_FRONT_AND_BACK`, um die verschiedenen Flächen einzustellen, sowie als `mode`:

- `GL_POINT` – es werden nur die Eckpunkte der gewünschten Fläche dargestellt
- `GL_LINE` – nur die Kanten der Fläche werden dargestellt: in der Anwendung entsteht dadurch ein Wireframe-Modell
- `GL_FILL` – der Standardmodus: die Fläche wird ausgefüllt dargestellt

Im Demoprogramm hat man die Möglichkeit, die Objekte als Wireframe darzustellen (4.6):

```
glPolygonMode(GL_FRONT_AND_BACK,e.line?GL_LINE:GL_FILL);
```

4.4.5 Koordinatensystem

```
if(axes) {
glDisable(GL_LIGHTING);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0,0,zscreen);
glBegin(GL_LINES);
glColor4f(1.,0.,0.,1.);
glVertex3f(-10.0,0.0,0.0);
glVertex3f(10.0,0.0,0.0);

glColor4f(0.,1.,0.,1.);
glVertex3f(0.0,-10.0,0.0);
glVertex3f(0.0,10.0,0.0);

glColor4f(0.,0.,1.,1.);
glVertex3f(0.0,0.0,znear);
glVertex3f(0.0,0.0,zfar);
glEnd();
```

```
glColor4fv(materialAmbientIntensity);
if(e.lighting) glEnable(GL_LIGHTING);
}
```

Falls der Benutzer das Achsenkreuz anzeigen möchte, wird zunächst die Belichtung deaktiviert, damit die Farben der Achsen nicht dadurch beeinflusst werden. Danach verschiebt man das Achsenkreuz auf die Konvergenzebene und wechselt dann in den `GL_LINES`-Modus, der bis zur `glEnd()`-Anweisung jeweils aus Start- und Endpunkt bestehende Strecken zeichnet. Die x-Achse wird rot, die y-Achse grün und die z-Achse blau gezeichnet.

Nach dem Zeichnen der Achsen setzt man durch `glColor4fv` die Farbe der folgenden Modelle auf die vorher definierte Materialfarbe. Das ist nur relevant, wenn die Belichtung deaktiviert wird, denn sonst kommen die durch `glColor*` gesetzten Farben nicht zum Einsatz, sondern die Farben und Intensitäten der Lichtquellen (definiert in Kapitel 4.1.3). Wenn man den letzten `glColor*`-Aufruf auskommentiert und die Belichtung deaktiviert ist, bleibt einfach die vorherige `glColor4f`-Anweisung in Kraft, also werden die dargestellten Modelle in der blauen Farbe der z-Achse gezeichnet.

Dieses Beispiel zeigt, wie einfach der Umgang mit unbelichteten Modellen sein kann.

4.4.6 Der stereoskopische Betrieb

```
if(stereo) {
    red:
        glClear(GL_DEPTH_BUFFER_BIT);
        glColorMask(1,0,0,1);
        a=0; eye=sep/2;
        goto render;
    blue:
        glClear(GL_DEPTH_BUFFER_BIT);
        glColorMask(0,1,1,1);
        a=1; eye=-sep/2;
        goto render;
}

render:
/* ... */

if(stereo) {
    if(a==0) goto blue;
    glColorMask(1,1,1,1);
}
```

Nach den anfänglichen Initialisierungen und Anpassungen an die vom Benutzer gewünschten Funktionen (denn die `display`-Prozedur wird ja nach jedem Redisplay aufgerufen [20]) kommen die Farbmasken zum Einsatz: falls man im stereokopischen Modus ist, wird zuerst das rote 'Auge' mit einer roten Maske gerendert und um $\frac{sep}{2}$ nach links verschoben und danach noch einmal die Szene mit einer cyanfarbenen Maske und nach rechts verschobenem Sichtpunkt dargestellt. Danach wird die Farbmaske wieder zurückgesetzt.

Falls man nur rote und blaue Objekte ohne Belichtung darstellt, kann man auch eine rote und eine blaue Maske verwenden, wobei eventuelle Grünanteile verworfen werden. Das kann etwa bei rot-blauen Anaglyphenbrillen notwendig werden; analog dazu kann man sich auf jede andere Variante von Anaglyphen einrichten, indem man einfach die Farbmasken anpasst.

Die OpenGL-Prozedur

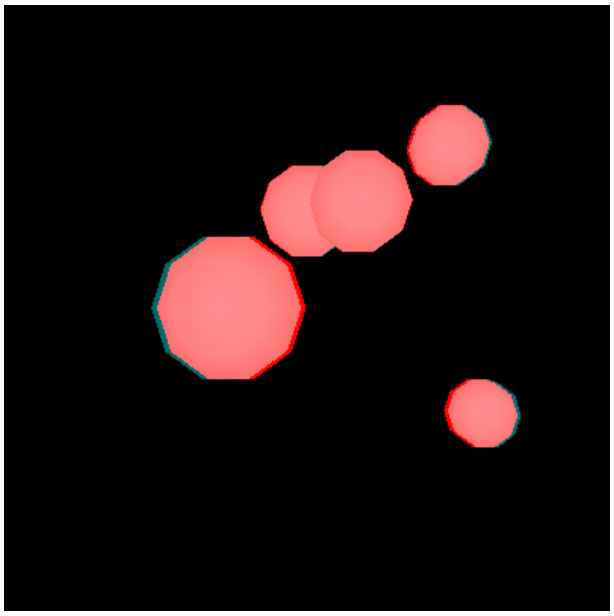


Abbildung 4.4: Stereoskopische Darstellung einer Szene

```
void glColorMask(GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha);
```

hat folgende Eigenschaften: [12]

- ein gesetztes Flag (`GL_TRUE`) bedeutet, dass die gewählte Komponente in den Farbpuffer geschrieben wird
- eine ausgeschaltete Komponente (`GL_FALSE`) wird nicht in den Farbpuffer geschrieben

So lassen sich sehr einfach bestimmte Komponenten maskieren.

Wie in 1.2.2 erläutert, werden die beiden Bilder überlagert und erzeugen durch die noch folgende Frustum-Projektion die Illusion der Räumlichkeit.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(...);
```

Man befindet sich nach der ersten Anweisung im Projektionsmodus, der speziell zur Anpassung der Perspektive dient. Die OpenGL-Prozedur `glMatrixMode` kann folgende Argumente annehmen: [11]

- `GL_PROJECTION` wechselt zur Projektionsmatrix, die in folgenden Anweisungen bearbeitet werden kann. Diese Matrix beschreibt die Weite des Sichtfeldes und die dargestellte Fläche, wie in Abbildung 1.13 erkennbar ist.
- `GL_MODELVIEW` wechselt zur Modelview-Matrix. Diese bestimmt die Transformationen, die auf Objekte in der Szene oder die Sicht auf diese angewendet werden.
- `GL_TEXTURE` wechselt zur Texturenmatrix. Diese wird hier nicht näher erläutert.

Die zweite Anweisung `glLoadIdentity` überschreibt die aktuell gewählte Matrix mit der Einheitsmatrix. Das ist vorteilhaft, da die meisten Transformationen aus Matrixmultiplikationen bestehen [11]. Weitere Informationen zu den OpenGL-Matrizen sind in Anhang A.4 zu finden.

4.4.7 Anpassen des Frustums

```
/* l1/(w/2-sep/2) = l2/(w/2+sep/2) = dnear/dscreen (für r1/2 analog)*/
glFrustum(-(w/2-eye)*dnear/dscreen,(w/2+eye)*dnear/dscreen,
    -dnear*tan(fov/2),dnear*tan(fov/2),
    dnear,dfar);
```

Die Prozedur `glFrustum` bestimmt nun die Abmessungen des Pyramidenstumpfes, der die Projektion modelliert. Dazu sei auf die Abbildung 1.14 sowie den Zweiten Strahlensatz in Kapitel 1.4.2 verwiesen.

Die Definition von `glFrustum` ist folgende: [11]

```
void glFrustum(GLdouble left, GLdouble right,
    GLdouble bottom, GLdouble top,
    GLdouble near, GLdouble far);
```

Die Prozedur definiert eine Perspektivmatrix. Die linke untere Ecke der Near Plane hat die Koordinaten $(left, bottom, -near)$, die rechte obere Ecke liegt bei $(right, top, -near)$. [11] Für $near$ sollte man also positive Werte verwenden, die dann in die negativen z-Koordinaten umgesetzt werden.

4.4.8 Rendern der Objekte

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(eye,0.0,0.0); /*Auge verschieben*/
glTranslatef(0,0,zscreen);
if(selector==CUBE) {
 /*...*/
} else if
 ... /* andere Modelle */
```

Zunächst wechselt man zur Modelview-Matrix und überschreibt diese mit der Einheitsmatrix. Dann führt man eine erste Sichttransformation (Viewing Transformation) durch: die Variable eye hat im stereoskopischen Betrieb den Wert $\pm\frac{sep}{2}$, also wird die Szene als letzte Operation um die Separation verschoben (Kapitel 1.2.3). Wegen der Matrixmultiplikation muss man darauf achten, die zuletzt durchzuführenden Transformationen zuerst zu codieren (Anhang A.4).

Davor soll die Szene um z_{screen} in Richtung der z-Achse verschoben werden: z_{screen} ist negativ, also wird alles Vorherige nach hinten verschoben.[1]

Schließlich entscheidet sich hier, welches Modell dargestellt wird; der Benutzer hat die Wahl zwischen verschiedenen Modellen, die später aufgelistet sind. Da die Verfahren ähnlich ablaufen, wird nur der allgemeine Ablauf erläutert:

- Ein Aufruf an `glPushMatrix()` kopiert die aktuelle Matrix auf den Stack.
- Es werden Modelltransformationen (Modelling Transformation) auf das zu rendernde Objekt angewendet. Hier ist es wieder relevant, ob zuerst eine Rotation oder eine Verschiebung stattfindet.[2]
- Durch eine GLUT-Hilfsfunktion wird das gewünschte Objekt gezeichnet.

[1] Zur Erinnerung: In OpenGL zeigt die Kamera standardmäßig vom Ursprung aus nach $-z$. [11]

[2] Würde etwa zuerst der Würfel verschoben und dann gedreht (so wie es hier *nicht* ist), würde sich der Würfel natürlich nicht auf der Stelle drehen, sondern auf einer 'Umlaufbahn' um den Ursprung – und zwar in dem Abstand, in dem er verschoben wurde.

- Die bearbeitete Matrix wird durch `glPopMatrix` verworfen.

Hier werden der Vollständigkeit halber die Befehle zum Zeichnen aller möglichen Modelle aufgeführt:

Modell 1: Würfel

Dieses sehr einfache Modell kann rotiert und in z-Richtung transformiert werden:

```
glPushMatrix();
glTranslatef(c.x, c.y, c.z);
glRotatef(c.angleX, 1.0, 0.0, 0.0);
glRotatef(c.angleY, 0.0, 1.0, 0.0);
glRotatef(c.angleZ, 0.0, 0.0, 1.0);
glutSolidCube(1.0);
glPopMatrix();
```

Modell 2: Kugeln

Die Kugeln befinden sich an verschiedenen Positionen, meist so, dass eine anschauliche Kollision entsteht:

```
for(i=0; i<NumBalls; i++) {
    glPushMatrix();
    glTranslatef(b[i].pos[X],b[i].pos[Y],b[i].pos[Z]);
    glutSolidSphere(1.0,10,5);
    glPopMatrix();
}
```

Modell 3: Bézierfläche

Durch einen Evaluator lässt sich eine Bézierfläche über die Variablen (u, v) darstellen, die im Wireframe-Modus sehr interessant aussieht. Diese kann man auch rotieren:

```
glPushMatrix();
glRotatef(cv.angle[X], 1.0, 0.0, 0.0);
glRotatef(cv.angle[Y], 0.0, 1.0, 0.0);
glRotatef(cv.angle[Z], 0.0, 0.0, 1.0);
glMap2f(GL_MAP2_VERTEX_3, /* evaluator will create 3D vertices */
    0.0,1.0, 3,3, /* u=[0;1], width=3, order=3*/
    0.0,1.0, 9,3, /* v=[0;1], width=9, order=3*/
    (float*)cv.c);
glEnable(GL_MAP2_VERTEX_3);
glMapGrid2f(50,0.0,1.0, /*u: in 50 steps from 0 to 1*/
    50,0.,1. /*v: in 50 steps from 0 to 1*/
);
glEvalMesh2(e.line?GL_LINE:GL_FILL,
    0,50,   /*evaluate u from step 0 to step 50*/
    0,50    /*evaluate v from step 0 to step 50*/
);
glPopMatrix();
```

Die Prozedur `glMap2f` erstellt eine Bézierfunktion dritter Ordnung mit den Kontrollpunkten aus `cv.c`.

Diese Funktion wird von OpenGL evaluiert: entweder durch mehrfaches Aufrufen von `glEvalCoord2*(u,v)` oder (hier) durch `glEvalMesh2`, welches über ein vorher mit `glMapGrid2f` definiertes Raster iteriert und für jeden Schritt den gewünschten Befehl emittiert.

In diesem Fall werden `glVertex3fv()`-Anweisungen generiert (wegen der Option `GL_MAP2_VERTEX_3`). Interessanterweise kann man auch andere OpenGL-Anweisungen durch eine Bézierfunktion berechnen – z.B. RGBA-Farbwerte oder Texturenkoordinaten [31].

Modell 4: Tesserakt und Pentachoron

Als Spezialität wurde ein vierdimensionaler „Hyperwürfel“ (Tesserakt) implementiert, der in das dreidimensionale Koordinatensystem projiziert wird. Außerdem gibt es als zweites vierdimensionales Modell ein Pentachoron (Simplex). Diese beiden Körper sind aus Kapitel 3 bekannt und werden ebenfalls in der Display-Prozedur dargestellt.

Im folgenden Code wurde der Vorgang angedeutet:

```
/* do rotations parallel to every plane:
    ...
*/
/* draw selected model: */
glBegin(GL_LINES);
for(i=0; i<h.numE; i++) {
    if(h.m==0) {
        x1=h.v[h.cubeE[i][0]][X]; y1=h.v[h.cubeE[i][0]][Y];
            z1=h.v[h.cubeE[i][0]][Z]; w1=h.v[h.cubeE[i][0]][W];
        x2=h.v[h.cubeE[i][1]][X]; /* ... */
    } else if(h.m==1) {
        x1=h.v[h.simplexE[i][0]][X]; /* ... */
        x2=h.v[h.simplexE[i][1]][X]; /* ... */
    }
    if(h.proj==0) {
        glVertex3f(x1,y1,z1);
        glVertex3f(x2,y2,z2);
    } else {
        /* perspective projection */
    }
}
```

Die Ecken und Kanten des Tesserakts wurden per Hand in die Datenstrukturen `h.cubeV` und `h.cubeE` eingetragen. Aus `h.cubeV` wurden schließlich nach Rotationen um 6 Ebenen die tatsächlichen Koordinaten erhalten und in `h.v` abgelegt. Nun wird entweder die drei- oder vierdimensionale `glVertex`-Funktion aufgerufen, woraufhin die Ecken und Kanten entweder in Parallelprojektion oder perspektivisch gezeichnet werden. Der obenstehende Code bezieht sich nur auf den Wireframe-Modus. Im Fill-Modus muss eine abgewandelte Fassung verwendet werden.

Die Details der Realisierung sind in Anhang B.1 zu finden.

4.4.9 Austauschen der Puffer

```
/*after render:*/
glutSwapBuffers();
```

Da das Program Double Buffering verwendet, werden am Ende der Display-Prozedur die Farbpuffer ausgetauscht. GLUT vereinfacht den Umgang mit Double Buffering so weit, dass man nur die oben genannte Hilfsprozedur aufrufen muss. Wie auf Seite 28 erwähnt, wurden bis zu diesem Zeitpunkt alle Operationen auf dem hinteren Farbpuffer (`GL_BACK`) durchgeführt. [12] Dieser wird nun in den vorderen Puffer geschrieben; die Inhalte des hinteren Puffers werden undefiniert [16].

Dadurch verhindert man Tearing, das zur ruckelnden Darstellung von Objekten führt, die zwischen zwei Frames 'abgehackt' werden:

> "When VSync is disabled in-game, screen tearing is observed when the frame rate exceeds the refresh rate of the display (120 frames per second on a 60Hz display, for example). This causes screen-wide horizontal tears whenever the camera or viewpoint moves horizontally or vertically." [18]

4.5 Die Keyboard-Prozedur

Das Keyboard-Callback ist sehr einfach zu realisieren: es genügt eine `switch-case`-Sequenz, die den eingehenden ASCII-Code überprüft. Hier soll nur beispielhaft das Vorgehen gezeigt werden:

```
void keyboard(unsigned char p1, int p2, int p3) {
    switch(p1) {
        case '1':
            selector=CUBE;
            break;
        case '2':
            selector=BALLS;
            break;
        case 'M':
            stereo=!stereo;
            break;
        case 'L':
            e.lighting=!e.lighting;
            break;
        /* ... */
    }
    glutPostRedisplay();

    printf("  Environment\n");
    printf("\tMode (M):\t\t%s\n",stereo?"STEREO":"MONO");
    printf(...);
    printf("\n");
}
```

Bei jeder Tastatureingabe in das GLUT-Fenster wird also gleichzeitig eine informative Nachricht über den Zustand des Systems in das Terminal geschrieben. Die weggelassenen Zeilen betreffen unter anderem Funktionen, die nur für eines der beiden möglichen Modelle zutreffen: so kann man den mit ASCII-Zeichen `1` wählbaren Würfel noch zusätzlich zu den globalen Optionen mit den Tasten `wasdqerf` rotieren und bewegen.

4.6 Die Menu-Prozedur

Als Alternative zur Tastatureingabe kann man einige der möglichen Optionen auch über das mit einem Rechtsklick zu öffnende Menü auswählen.

```
void menu(int p) {
    switch(p) {
        case 0:
            e.lighting=!e.lighting;
            break;
        /* ... */
    }
    keyboard('#',-1,-1);
    glutPostRedisplay();
}
```

Die simple Prozedur prüft den erhaltenen Menü-Code (der in `main` einem Eintrag zugewiesen wurde) und schaltet die entsprechende globale Variable um. Danach wird ein Dummy-Aufruf an das Keyboard-Callback getätigt, um auch hier einen informativen Text auf das Terminal zu schreiben. Als letztes signalisiert man dem GLUT-System, dass das Grafikfenster neu gezeichnet werden soll.

4.7 Die Reshape-Prozedur

```
width=p1;
height=p2;
glViewport(0,0,width,height);
```

Zunächst stellt man die globalen Variablen für Fensterbreite und -höhe auf das erste und zweite Argument ein. Daraus erhält man später das Seitenverhältnis, das zur Berechnung der virtuellen Bildschirmbreite in 4.4 benötigt wird.

Die OpenGL-Prozedur `glViewport` stellt den Viewport ein: Die Viewport-Transformation bestimmt, in welchem Bereich die grafische Szene auf dem Grafikfenster dargestellt wird. Hier soll die Szene das Fenster vollständig ausfüllen.

4.8 Die Timer-Prozedur

```
if(selector==CUBE&&e.anim) {
    c.angleY+=1.0;
    while(c.angleY>360) c.angleY-=360.0;
}
```

Hier wird überprüft, ob der Benutzer den Würfel automatisch rotieren lassen möchte (durch Auswahl des entsprechenden Menüpunktes). Bei jedem Aufruf der `timer`-Funktion, also alle 5 ms, wird der Würfel ein Stück

weiter in y-Richtung rotiert.

```
else if(selector==BALLS&&e.anim) {
    for(a1=1; a1<NumBalls; a1++) {
        for(a2=0; a2<a1; a2++) {
            if(near(b[a1].pos[X],b[a2].pos[X]) &&
                near(b[a1].pos[Y],b[a2].pos[Y]) &&
                near(b[a1].pos[Z],b[a2].pos[Z])) {
                /* Kollision findet statt */
            }
        }
    }
    for(i=0; i<NumBalls; i++) {
        b1x=b[i].v[X]; b1y=b[i].v[Y]; b1z=b[i].v[Z];
        if(!near(b1x,0.0,0.01)) b[i].v[X]=b1x>0?
            b1x-cd*(b1x*b1x)/2:b1x+cd*(b1x*b1x)/2;
        /* analog für b1y und by1z */

        b[i].pos[X]+=b[i].v[X];
        b[i].pos[Y]+=b[i].v[Y];
        b[i].pos[Z]+=b[i].v[Z];
    }
}
glutPostRedisplay();
glutTimerFunc(5,timer,0);
```

Hier wird nun das System aus kollisionsfähigen Kugeln zusammengestellt. Zuerst wird, falls die Animation aktiviert ist, auf Kollision zweier Kugeln überprüft. Danach kommt der simulierte Luftwiderstand zum Einsatz, der die Geschwindigkeiten ständig verringert. Zuletzt wird jede Kugel um den Betrag ihrer Geschwindigkeit verschoben.

Der Timer bewirkt, dass dieser Vorgang alle $5\mathrm{ms}$ durchgeführt wird.

4.8.1 Kollisionserkennung

Zur Erkennung einer Kollision iteriert man über die Objekte $b_k,\ 1 \leq k \leq n$ und $b_i,\ 0 \leq i < k$. Für jedes b_i wird nun überprüft, ob alle drei Komponenten des Positionsvektors in der Nähe eines anderen Objekts b_k liegen: $\vec{b_i} \approx \vec{b_k}$. Dazu wurden die Hilfsprozeduren `absd` und `near` erstellt:

```
double absd(double p) { return p<0?-p:p; }
double near(double p1,double p2) { return absd(p1-p2)<1.0; }
```

Wenn der Betrag aller drei Komponenten also jeweils kleiner als 1 ist, wird für beide Körper eine Kollision simuliert. Der Radius einer Kugel ist genau 1 Einheit, da dieser Wert in der Display-Prozedur zum Darstellen der Kugeln (4.4, `glutSolidSphere`) verwendet wurde.

4.8.2 Teilelastische Kollision

```
/* m1=0.5, m2=0.5 */
b1x=b[a1].v[X]; b1y=b[a1].v[Y]; b1z=b[a1].v[Z];
b2x=b[a2].v[X]; b2y=b[a2].v[Y]; b2z=b[a2].v[Z];
b[a1].v[X]=b1x*0.5+b2x*0.5+0.5*cr*(b2x-b1x);
b[a1].v[Y]=b1y*0.5+b2y*0.5+0.5*cr*(b2y-b1y);
b[a1].v[Z]=b1z*0.5+b2z*0.5+0.5*cr*(b2z-b1z);
b[a2].v[X]=b1x*0.5+b2x*0.5+0.5*cr*(b1x-b2x);
b[a2].v[Y]=b1y*0.5+b2y*0.5+0.5*cr*(b1y-b2y);
b[a2].v[Z]=b1z*0.5+b2z*0.5+0.5*cr*(b1z-b2z);
```

Bei einer Kollision von `b[a1]` mit `b[a2]` werden die beiden Geschwindigkeiten nach Gleichung 2.1 verändert.

4.8.3 Reibung

Im Demoprogramm gibt es einen Mechanismus für die Simulation von Reibung oder Luftwiderstand. In periodischen Abständen werden die Geschwindigkeiten um einen bestimmten Anteil dem Stillstand angenähert, sodass sich die Kugeln annähernd so verhalten wie Billardkugeln auf einem sehr reibungsarmen Tisch. [3]

Dazu wurde die bekannte Formel für den Luftwiderstand verwendet (Gleichung 2.2). Im Demoprogramm wurde der Parameter `cd` experimentell auf etwa $0,2$ oder $0,3$ festgelegt.

---

[3] Tatsächlich haben Billardkugeln einen Restitutionskoeffizienten von $e > 1$, da sie mit einer Cellulosenitratbeschichtung versehen sind. Bei dieser superelastischen Kollision wird zusätzliche Energie aus der chemischen Reaktion (einer kleinen Explosion) gewonnen, wodurch die Billardkugel beschleunigt wird.

Kapitel 5

Fazit

5.1 Erkenntnisse

Während der Bearbeitung des Projekts habe ich folgende Erkenntnisse gewonnen:

- Grundlagen der Stereoskopie: Vor dem Projekt war ich auf diesem Gebiet vollkommen unwissend, und in mühevoller Arbeit und mit Hilfe von [5] und [2] konnte ich schließlich die vorgestellten Konzepte zuverlässig in der Anwendung implementieren. Besonders für diese Implementierung waren [7] sowie [4] eine wertvolle Hilfe.
- Umgang mit OpenGL und GLUT: Auch wenn ich schon zuvor mit OpenGL gearbeitet habe, war es dennoch aufwendig, die Eigenheiten und Konzepte (wieder) zu verstehen. Das OpenGL Utility Toolkit war natürlich eine große Hilfe beim Aufbau des Demoprogramms, da das GLUT-System dem Entwickler u.a. die Aufgaben des Event Handling abnimmt. Hier waren wertvolle Quellen [11] und [20].
- Besserer Umgang mit LaTeX: Das Textprogramm LaTeX bietet ausgezeichnete Funktionen für wissenschaftliche Arbeiten, jedoch ist die Bedienung gewöhnungsbedürftig. Nach der Bearbeitung dieses technischen Berichts kann ich recht gut die wichtigsten Funktionen einsetzen, und habe außerdem noch ein Anschauungsobjekt erstellt, an dem ich für spätere Berichte die Konstruktion einer solchen Arbeit direkt nachvollziehen kann.
- Erstellen von Skizzen mit TikZ: Eine sehr neue erworbene Fähigkeit ist das Zeichnen von Graphen (wie Abbildung 1.14) mit dem in LaTeX integrierten Zeichenprogramm TikZ. Auch dieses Programm ist anfangs nicht einfach zu bedienen, doch ich habe das nötigste aus der Dokumentation entnehmen können.
- Physikalische Grundlagen: Die Arbeit an der Kollisionserkennung und -durchführung hat mich durch viele Internetquellen geführt, von denen [21] eine der besseren war. Im Code sind noch die ersten Versionen der Kollision zu finden, die jedoch nicht korrekt funktionierten.
- Vierdimensionale Abbildung: Nach der Lektüre von Paul Bourkes Artikeln zum Hyperraum war ich sehr daran interessiert, eine dreidimensionale Darstellung vierdimensionaler Objekte zu realisieren. Aber ich musste schnell feststellen, dass sich das nicht „einfach so“ umsetzen lässt. Besonders die Implementierung der 4D-Modelle war eine außergewöhnlich schwierige Aufgabe, da [34] sich doch sehr kurz gefasst hat. Aber nach einigen Stunden harter Arbeit ist ein ausgezeichnetes Ergebnis entstanden.
- Geduld: Es gilt wohl für alle der oben genannten Themen, dass man sich intensiv in jedes einarbeiten muss und auch eine Menge von Fehlern oder unerwartetem Verhalten beheben muss. Das beste Beispiel ist die

Arbeit mit `glFrustum`, die einen ganzen Komplex von technischen und mathematischen Problemen – Near Plane, virtueller Bildschirm, Zweiter Strahlensatz, asymmetrische Projektion, vertikaler Bildwinkel, und nicht zu vergessen: negative z-Koordinaten in OpenGL – mit sich bringt.

5.2 Mögliche Erweiterungen

Während der Arbeit an diesem Bericht habe ich zwangsläufig jede Zeile des Demoprogramms noch einmal untersucht. Wegen der gleichzeitigen Lektüre der OpenGL- und GLUT-Spezifikationen sind mir diese Funktionalitäten aufgefallen, die eventuell noch implementiert werden könnten:

- Verwendung einer `GL_STEREO`-Umgebung: Wie in 4.4 genannt, verwendet das Programm den `GL_BACK`- und den `GL_FRONT`-Farbpuffer. Beim Lesen von [12] fiel mir die Erwähnung von Stereopaaren von Puffern auf: die Puffer `GL_BACK_LEFT` und `GL_BACK_RIGHT` sowie die korrespondierenden vorderen Puffer können auch zur Erzeugung von stereoskopischen Anwendungen gebraucht werden. Aber ich habe mich noch nicht mit diesem Mechanismus beschäftigt und meine außerdem, dass die Verwendung von `glColorMask` sehr elegant und nicht ressourcenintensiver als die oben genannte Technik ist. Deshalb werde ich diese, wenn überhaupt, für eventuelle spätere Projekte in Betracht ziehen.
- Bewegliches Koordinatensystem: Ein Wechsel zur Transformation des Koordinatensystems wie in [4] wäre zwar denkbar, aber meiner Meinung nach nicht notwendig. Es gibt dennoch die Möglichkeit, mit dem Kontextmenü die (unbeweglichen) Achsen anzuzeigen.
- Wählbare Separation: Das Einstellen der Separation könnte durch den Benutzer geschehen, da verschiedene Betrachter aufgrund ihrer Physiologie durchaus unterschiedliche Parallaxbereiche benötigen [2].
- Depth Cueing in der vierten Dimension: Es könnte noch sinnvoll sein, weiter 'hinten' liegende Kanten anders einzufärben, also abhängig von der w-Koordinate. [33]

5.3 Zusammenfassung

Anfangs war ich mir noch nicht sicher, wie weit das Projekt gehen würde, da ich die Basiskonzepte der Stereoskopie erst durch die Anwendung im Programm zu verstehen begann. Aber besonders durch die Arbeit an diesem vorliegenden Bericht ist mir das Themengebiet viel klarer geworden, sodass ich sogar durch die Implementierung von einer Bézierfläche und vierdimensionalen Objekten in verschiedene Richtungen experimentieren konnte. Dabei war die Möglichkeit der stereoskopischen Betrachtung nur ein Bonus, der die erzielten grafischen Ergebnisse noch interessanter machte.

Nicht nur auf dem Gebiet der Stereoskopie, sondern auch der Computergrafik, bin ich durch die Arbeit an diesem Projekt deutlich fortgeschritten. Besonders die Tesseraktmodellierung war eine große Herausforderung, die sich fast zum Kernstück des Berichts entwickelt hat.

Ich hoffe, dass ich mit diesem Bericht eine genügende Menge an Informationen gesammelt habe, um mir spätere Arbeiten mit OpenGL und GLUT zu erleichtern. Insgesamt bin ich zufrieden mit dem Aufbau des Demoprogramms, dass sich zu einer Art Schaukasten für verschiedene stereoskopische Modelle entwickelt hat, zwischen denen man einfach umherschalten kann. Dabei hat jedes Modell einige spezielle Funktionen, die der Betrachter erforschen kann.

Anhang A

Weiterführende Informationen

Hier werden Konzepte erläutert, die zwar im Bericht angewendet werden, aber entweder oft schon bekannt oder nur zu einem kleinen Teil relevant sind. Daher wurden diese Theorien in diesen Anhang verlagert, wo man sie bei Interesse nachlesen kann.

A.1 Sichtebenen

Man verwendet in binokularen Anwendungen meist drei Sichtebenen, die in bestimmter Entferunung zum Betrachter liegen.

Die z-Koordinate ist eine Angabe für die Entfernung eines Objekts vom Augenpunkt. Im OpenGL-Koordinatensystem gilt $z < 0$: je kleiner die z-Koordinate ist, desto weiter entfernt ist ein Objekt. Denn am Anfang eines OpenGL-Programms befindet sich die 'Kamera' am Ursprung und schaut in $-z$-Richtung [11].

Near Plane Die 'nahe Ebene' oder Near Plane liegt bei z_{near}. Objekte, die bei z_{near} liegen, befinden sich direkt vor den Augen (dem Auge) des Betrachters. Ein Objekt, das sich noch näher an den Augenpunkt anzunähern versucht, wird abgeschnitten – daher auch die Bezeichnung *Near Clipping Plane*[11]. In einer Anwendung mit monokularer Perspektive wird die Szene durch einen Viewport auf die Near Plane abgebildet, es gibt nicht die Notwendigkeit der Einstellung auf die verschiedenen Sichtlinien. [2]

Far Plane Analog dazu stellt die Far Plane den am weitesten entfernten Punkt dar, an dem Objekte noch dargestellt werden. Dahinter liegende Objekte werden ebenfalls abgeschnitten. Die Konstante z_{far} wird meist sehr hoch gewählt, sodass nicht fälschlicherweise geclippt wird. [2]

Virtual Screen Der virtuelle Bildschirm ist in der binokularen Anwendung die Ebene, bei der die beiden Sichtkegel konvergieren (Konvergenzebene). Dazu wird eine Konstante z_{screen} definiert, die die Translation des Screens weg vom Betrachter beschreibt. Vom virtuellen Bildschirm aus empfangen die Augen die Szene horizontal versetzt, und zwar in dem Ausmaß der *Separation*. [2]

A.2 Analyse: Das OpenGL-Lichtsystem

In OpenGL werden Lichtquellen durch die Anteile an rotem, grünem, und blauem Licht charakterisiert, das sie aussenden. [10]

- Umgebungslicht (`GL_AMBIENT`) ist die Grundbelichtung der Szene, vergleichbar mit einem professionell ausgeleuchteten Filmset. Das Licht hat keinen bestimmbaren Ursprungsort.
- Diffuses Licht (`GL_DIFFUSE`) ist gerichtetes, großflächiges Licht, das beim Beleuchten einer Oberfläche eher gleichmäßig reflektiert wird. Ein gedämpfter Scheinwerfer produziert ein diffuses Licht, das auch Schatten entstehen lassen kann, jedoch meist sehr weich ist.
- Gespiegeltes Licht (`GL_SPECULAR`) ist der Anteil des Lichts, der von spiegelnden Oberflächen sehr gezielt reflektiert wird. Als Beispiel gilt etwa das Licht einer Taschenlampe.

Analog dazu gilt für Oberflächen von Objekten, dass sie einen bestimmten Anteil eingehenden Lichts reflektieren, der durch die Vektoren angegeben wird. Im Beispiel reflektiert das Material eines Objekts einen bestimmten Anteil des Umgebungslichtes (`GL_AMBIENT`). Dazu könnte man auch die Reflektionsstärke für diffuse und 'spiegelnde' Lichtquellen definieren.

Eine Besonderheit der `glMaterial*(...)`-Prozedur ist der Parameter `GL_SHININESS`, mit dem sich das Ausmaß der Spiegelung ("specular reflection" [10]) bestimmen lässt.

Die Intensitätsvektoren Bei der Bestimmung einer Lichtquelle geben die Vektoren die Farbanteile des ausgestrahlten Lichts an, also ist ein Licht mit `{1.0,0.0,0.0}` ein reines, intensives Rot. – Bei der Bestimmung der Materialeigenschaften bestimmt der Vektor, welcher Anteil der jeweiligen Komponente des eingehenden Lichts reflektiert wird. Also wäre ein Material mit `{0.5,0.5,0.5}` eine Oberfläche, die rotes, grünes und blaues Licht zur Hälfte reflektiert, also eine eher matte weiße Fläche.

Die Positionsvektoren Wie oben beschrieben, sendet bei monokularer Anwendung nur eine Lichtquelle weißes Licht einer bestimmten Intensität aus. Die Position der Lichtquelle ist definiert als ein Vektor `{x,y,z,w}`[10]. Dabei bestimmt die `w`-Komponente, ob es sich um ein direktionales (`w=0.0`) oder positionales (`w=1.0`) Licht handelt.

- **Direktionales Licht** Von einer Lichtquelle ausgesandtes Licht, die unendlich weit entfernt simuliert wird. Die Koordinaten x, y, und z spezifizieren die Richtung der Lichtstrahlen entlang der jeweiligen Achse. [10]

- **Positionales Licht** Licht, dessen Ursprung sich im kartesischen Raum der Simulation befindet. In diesem Fall bestimmen die drei Koordinaten nicht die Richtung der Lichtstrahlen, sondern die Position der Lichtquelle. Folglich lässt sich die Richtung nicht ohne weiteres (das hier nicht behandelt wird) anpassen: die Lichtquelle strahlt in alle Richtungen.

A.3 Leeren der OpenGL-Puffer

Die Puffer sind zusammenhängende Speicherfelder, in denen verschiedene Daten festgehalten werden. Eine häufig durchgeführte Operation ist das Leeren eines Puffers.

Die Prozedur `void glClear(GLbitfield mask)` hat als Bitmaske folgende Optionen:

- `GL_COLOR_BUFFER_BIT` – aktiviert das Leeren des Farbpuffers (Color Buffer), der einfach den Speicherbereich darstellt, in dem alle Pixel der Anwendung sich befinden. Die Pixel bestehen hier aus vier Komponenten (R, G, B und A). [11]

- `GL_DEPTH_BUFFER_BIT` – leert den Depth Buffer, auch z-Puffer genannt, da dieser für jeden Pixel einen Tiefenwert speichert. Standardmäßig überschreiben Pixel mit niedrigeren z – also einer geringeren Entfernung zum Auge – solche mit höheren z. Mittels des *Depth Test* lässt sich dieses Verhalten aber verändern. [12]
- `GL_STENCIL_BUFFER_BIT` – leert den Stencil Buffer, der es ermöglicht, Bildschirmausschnitte zu maskieren. So kann man etwa ein Sichtfenster in den Stencil Buffer schreiben, und alles, das durch dieses Fenster sichtbar ist, wird dargestellt; der Rest wird nicht modifiziert. Das ermöglicht statische Bereiche (z.B. Cockpit im Flugsimulator), die nicht ständig erneuert werden müssen. [12]
- `GL_ACCUM_BUFFER_BIT` – leert den Accumulation Buffer, der zum Sammeln von Bildern geeignet ist, die dann etwa durch Supersampling überlagert werden. [12] Man kann den Accumulation Buffer ebenfalls zum Erzeugen von stereoskopischen Anwendungen verwenden, aber die hier verwendeten Farbmasken sind speichereffizienter und ebenso einfach zu verstehen.

A.4 Analyse: Die OpenGL-Matrizen

Zum Umgang mit den Matrizen, die OpenGL für die Speicherung von Transformationen verwendet, gibt es vier wichtige Funktionalitäten:

- Wechseln der bearbeiteten Matrix mit `glMatrixMode`
- Laden der Einheitsmatrix mit `glLoadIdentity`
- Transformationen der Szene mit `glTranslatef`, `glScalef` oder `glRotatef` (im Modelview-Modus)
- Interaktion mit dem Matrixstack: `glPushMatrix`, `glPopMatrix`

Im Projektionsmodus (`GL_PROJECTION`) wird in Kapitel 4.4.6 das Frustum angepasst. Im Modelview-Modus wird die Konfiguration der Szene verändert:

Transformationen im Modelview-Modus Der Modelview-Modus (`GL_MODELVIEW`) hat seinen Namen durch Zusammensetzen von *Modelling Transformation* und *Viewing Transformation* erhalten. Der Grund dafür ist die ähnliche Verwendbarkeit der Transformationsfunktionen für Objekte oder die 'Kamera', die auf diese Objekte gerichtet ist. [11] Die oben genannten Transformationsprozeduren operieren alle auf der Modelview-Matrix, also ist nur die Reihenfolge der ausgeführten Operationen von Bedeutung:
Wenn man nach Laden der Einheitsmatrix zuerst die Szene mit `glTranslatef(0,0,1.0)` um 1 Einheit nach vorne verschiebt und dann ein Objekt zeichnet, werden folgende Transformationen des Objekts sich natürlich auf diese Matrix beziehen, was ununterscheidbar von einer Änderung des Sichtpunktes ist. [11]

Alle bisher beschriebenen Funktionen operieren auf der *aktuellen Matrix* – nämlich auf dem obersten Element des Matrixstacks [11]:

Der Matrixstack Der Stack dient zum Erhalten von früheren Konfigurationen, sodass man aufwendige Initialisierungen nicht immer wieder durchführen muss:

> "Since the transformations are stored as matrices, a matrix stack provides an ideal mechanism for doing this sort of successive remembering, translating, and throwing away." [11]

Nach dem bekannten Stackmechanismus legt die Prozedur `glPushMatrix()` eine Kopie der aktuellen Matrix auf den Stack, und `glPopMatrix()` verwirft die aktuelle Matrix. Falls man den Matrixstack nicht verwendet, operieren alle Transformationen auf einer einzigen Matrix.

Matrizenmultiplikation bei Transformationen Es ist bekannt, dass am Anfang die Einheitsmatrix die aktuelle Matrix ist. Wird nun darauf eine Transformation angewendet – also die beiden Matrizen miteinander multipliziert –, ist die daraus entstehende Matrix die neue aktuelle Matrix (*Komposittransformation*).

$$M \cdot M_T = M'$$

Anschließend gezeichnete Objekte beziehen sich auf M', also würde ein Punkt p folgendermaßen verändert: [11]

$$p_{eff} = M' \cdot p = M \cdot M_T \cdot p$$

Bei mehreren sukzessiven Transformationen $M_{0..4}$ wäre das Ergebnis das folgende:

$$p_{eff} = M' \cdot p = M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot p$$

Achtung: Da die Matrizenmultiplikation nicht kommutativ ist, darf die Reihenfolge der Matrizen $M_{0..4}$ nicht verändert werden. [14] Daraus folgt: Es wird zuerst M_4 auf p angewendet, dann M_3 und so fort (oder eine in dieser Reihenfolge zusammengesetzte Matrix, was laut der Assoziativität erlaubt ist [14]). Das bedeutet, dass die zuerst auf ein Objekt angewandte Transformation in einer Anweisungsfolge als letzte stehen muss. [11]

Anhang B

Details der Implementierung

In diesem Anhang werden weitere Details der Umsetzung in OpenGL vorgestellt. Besonders die Umsetzung der Hyperspace-Modelle ist hier zu beachten.

B.1 Darstellen der vierdimensionalen Modelle

In Kapitel 3 wurden die Ideen der vierdimensionalen Abbildung mathematisch erklärt; in diesem Abschnitt folgt die Realisierung in OpenGL.

B.1.1 Standardkoordinaten der Modelle

Der Tesserakt und das Pentachoron benötigen beide jeweils drei globale Datenstrukturen zum Bestimmen ihrer Konfiguration:

- `float cubeV[16][4]` / `float simplexV[5][4]` – enthält die Koordinaten aller Eckpunkte
- `int cubeE[32][2]` / `int simplexE[10][2]` – enthält je zwei Indizes zur Identifikation einer Kante von A nach B. Diese Datenstruktur wird für die Darstellung im Wireframe-Modus benötigt.
- `int cubeF[24][4]` / `int simplexF[10][3]` – enthält vier/drei Indizes, um eine Fläche zu bestimmen. Wichtig ist, dass man den Drehsinn einer Fläche beachtet, also nur in einer Richtung um diese herumgeht, sonst entsteht eine falsche Form. Diese Datenstruktur wird nur für die Darstellung im Fill-Modus benötigt.

Der Standard-Tesserakt $(\pm 1, \pm 1, \pm 1, \pm 1)$ wird mit diesen Koordinaten initialisiert:

```
{
    /* hypercube vertices */
    {-1,-1,1,-1},{1,-1,1,-1},{-1,1,1,-1},{1,1,1,-1},
    {-1,-1,-1,-1},{1,-1,-1,-1},{-1,1,-1,-1},{1,1,-1,-1},
    {-1,-1,1,1},{1,-1,1,1},{-1,1,1,1},{1,1,1,1},
    {-1,-1,-1,1},{1,-1,-1,1},{-1,1,-1,1},{1,1,-1,1}
},
{
    /* hypercube edges */
    {0,1},{0,2},{0,4},{0,8}, {1,3},{1,5},{1,9}, {2,3},{2,6},{2,10}, {3,7},{3,11},
```

```
    {4,5},{4,6},{4,12}, {5,7},{5,13}, {6,7},{6,14}, {7,15},
    {8,9},{8,10},{8,12}, {9,11},{9,13}, {10,11},{10,14}, {11,15},
    {12,13},{12,14}, {13,15}, {14,15}
},
{
    /* hypercube faces */
    {0,1,3,2},{0,1,5,4},{0,2,6,4}, {0,1,9,8},{0,2,10,8},{0,4,12,8},
    {1,3,7,5}, {1,3,11,9},{1,5,13,9},
    {2,3,7,6}, {2,3,11,10},{2,6,14,10},
     {3,7,15,11},
    {4,5,7,6}, {4,5,13,12},{4,6,14,12},
     {5,7,15,13},
     {6,7,15,14},
    {8,9,11,10},{8,9,13,12},{8,10,14,12},
    {9,11,15,13},
    {10,11,15,14},
    {12,13,15,14}
}
```

Das Standard-Pentachoron[1] (Kantenlänge 2) wird mit diesen Koordinaten initialisiert:

```
{ /* simplex vertices */
    {-1,-1,-1,-1},{1,-1,-1,-1},{0,0.732,-1,-1},
    {0,-0.423,0.732,-1},{0,-0.423,-0.567,0.732}
},
{ /* simplex edges */
    {0,1},{0,2},{0,3},{0,4}, {1,2},{1,3},{1,4}, {2,3},{2,4}, {3,4}
},
{ /* simplex faces */
    {0,1,2},{0,1,3},{0,1,4},{0,2,3},{0,2,4},{0,3,4}, {1,2,3},{1,2,4},{1,3,4}, {2,3,4}
}
```

Diese oben gezeigten Arrays werden im Programm in die Struktur `h` geladen, wo sich auch die drei Zähler `numV`, `numE` und `numF` befinden. Diese spezifizieren die Anzahl der Ecken, Kanten und Flächen eines Modells. Beim Umschalten der Modelle (in der `keyboard`-Prozedur) müssen diese Zähler angepasst werden, da sonst über die Grenzen der Datenstrukturen hinaus iteriert wird.

Diese drei Datenstrukturen werden ansonsten nicht modifiziert. Rotationen geschehen auf der intermediären Datenstruktur `h.v`.

B.1.2 Rotation der Standardpunkte

Wir befinden uns nun in der `display`-Prozedur. Falls der Benutzer das Modell `4` ausgewählt hat, wird das Hyperspace-Modell geladen. Der Ablauf ist dann der folgende:

- Laden der Eckkoordinaten des Modells
- Rotieren des Modells um alle sechs Ebenen
- Darstellen des Modells

[1] Zur Erinnerung: Das Pentachoron wird auch (4-)Simplex genannt.

- Im Wireframe-Modus: Kanten des Modells auslesen und anzeigen
- Im Fill-Modus: Flächen des Modells auslesen und anzeigen (für den Tesserakt: quadratische Flächen, für das Pentachoron: dreieckige Flächen)

Hier wird zunächst das Behandeln der Rotation um die sechs Ebenen gezeigt:

```
for(i=0; i<h.numV; i++) {
    /* 6 Rotationen, jeweils alternierend x1/x2, y1/y2 usw. verwenden */
    if(h.m==0) {
        x2=h.cubeV[i][X];
        y2=h.cubeV[i][Y];
        z2=h.cubeV[i][Z];
        w2=h.cubeV[i][W];
    } else if(h.m==1) {
        x2=h.simplexV[i][X];
        y2=h.simplexV[i][Y];
        z2=h.simplexV[i][Z];
        w2=h.simplexV[i][W];
    }
    /* rotation around xy plane: */
    x1=(cos(h.angle[XY])*x2+sin(h.angle[XY])*y2);
    y1=(-sin(h.angle[XY])*x2+cos(h.angle[XY])*y2);
    z1=z2;
    w1=w2;
    /* rotation around yz plane: */
    y2=(cos(h.angle[YZ])*y1+sin(h.angle[YZ])*z1);
    z2=(-sin(h.angle[YZ]*y1)+cos(h.angle[YZ])*z1);
    x2=x1;
    w2=w1;
    /* rotation around xz plane: */
    x1=(cos(h.angle[XZ])*x2-sin(h.angle[XZ])*z2);
    z1=(sin(h.angle[XZ])*x2+cos(h.angle[XZ])*z2);
    y1=y2;
    w1=w2;

    /* rotation around xw plane: */
    x2=(cos(h.angle[XW])*x1+sin(h.angle[XW])*w1);
    w2=(-sin(h.angle[XW])*x1+cos(h.angle[XW])*w1);
    y2=y1;
    z2=z1;
    /* rotation around yw plane: */
    y1=(cos(h.angle[YW])*y2-sin(h.angle[YW])*w2);
    w1=(sin(h.angle[YW])*y2+cos(h.angle[YW])*w2);
    x1=x2;
    z1=z2;
    /* rotation around zw plane: */
    z2=(cos(h.angle[ZW])*z1-sin(h.angle[ZW])*w1);
    w2=(sin(h.angle[ZW])*z1+cos(h.angle[ZW])*w1);
    x2=x1;
    y2=y1;
```

```
        h.v[i][X]=x2; h.v[i][Y]=y2; h.v[i][Z]=z2; h.v[i][W]=w2;
    }
```

Zunächst entscheidet man, aus welchem Modell man die Koordinaten des aktuellen Punktes entnimmt: momentan sind der Tesserakt und das Pentachoron verfügbar. Die vier Komponenten werden dann in vier Variablen gelesen.

Im Folgenden werden zwei Variablengruppen immer alternierend belegt, sodass Änderungen bis zum Ende erhalten bleiben. Wie in 3.1.4 gezeigt, werden hier die Komponenten entsprechend der Matrizenmultiplikation verändert.

Am Ende werden die entstandenen Koordinaten des aktuellen Punktes in ein Feld `v` abgelegt, auf das dann die Darstellungsfunktionen zugreifen werden.

B.1.3 Darstellen der Kanten im Wireframe-Modus

Falls der Wireframe-Modus aktiviert ist, wird der folgende Code zum Darstellen verwendet. Falls nicht, werden die Flächen mit dem Code aus B.1.4 gezeichnet.

```
glBegin(GL_LINES);
for(i=0; i<h.numE; i++) {
    /* draw edges */
    if(h.m==0) {
        x1=h.v[h.cubeE[i][0]][X]; y1=h.v[h.cubeE[i][0]][Y]; z1=h.v[h.cubeE[i][0]][Z];
            w1=h.v[h.cubeE[i][0]][W];
        x2=h.v[h.cubeE[i][1]][X]; y2=h.v[h.cubeE[i][1]][Y]; z2=h.v[h.cubeE[i][1]][Z];
            w2=h.v[h.cubeE[i][1]][W];
    } else if(h.m==1) {
        x1=h.v[h.simplexE[i][0]][X]; y1=h.v[h.simplexE[i][0]][Y];
            z1=h.v[h.simplexE[i][0]][Z]; w1=h.v[h.simplexE[i][0]][W];
        x2=h.v[h.simplexE[i][1]][X]; y2=h.v[h.simplexE[i][1]][Y];
            z2=h.v[h.simplexE[i][1]][Z]; w2=h.v[h.simplexE[i][1]][W];
    }
    if(h.proj==0) {
        glVertex3f(x1,y1,z1);
        glVertex3f(x2,y2,z2);
    } else {
        /* hollasch: clip edges with negative w signs */
        if(w1>0.001&&w2>0.001) {
            glVertex4f(x1,y1,z1,w1);
            glVertex4f(x2,y2,z2,w2);
        } else if(w1>0.001||w2>0.001) {
            /*project to w=0 hyperplane*/
            mu=(0.001-w1)/(w2-w1);
            x3=(1-mu)*x1+mu*x2; y3=(1-mu)*y1+mu*y2; z3=(1-mu)*z1+mu*z2; w3=0.001;
            if(w1>0.001) {
                glVertex4f(x1,y1,z1,w1);
                glVertex4f(x3,y3,z3,w3);
            } else {
                glVertex4f(x3,y3,z3,w3);
                glVertex4f(x2,y2,z2,w2);
            }
```

```
            }
        }
    }
    glEnd();
```

Wieder wird zuerst das passende Modell ausgewählt. Diesmal sucht man die Kanten, die zwischen den Ecken des Modells aufgespannt sind, und speichert die Koordinaten beider Eckpunkte in `x1`, `y1`, `z1`, `w1` usw.

Falls die Parallelprojektion gewählt wurde, wird die w-Koordinate einfach verworfen. Das geschieht durch die Verwendung von `glVertex3f`.

Die Perspektivenprojektion ist deutlich aufwendiger: Zunächst werden alle Kanten mit beiden $w > 0$ zugelassen. Außerdem werden auch Kanten mit einem $w > 0$ zugelassen, nämlich indem die Ecke mit negativem w auf die $w = 0$-Hyperebene abgebildet wird. Dieser Vorgang wurde in 3.2.2 beschrieben. Dort steht auch, warum die entsprechende Ecke tatsächlich nach $w = 0,001$ projiziert wird. Hier werden die Variablen `x3` bis `z3` verwendet, um den Projektionspunkt zu erfassen. Dann werden die Koordinaten der 'negativen Ecke' durch die Koordinaten des Projektionspunktes ersetzt (der Punkt wird gegen die w-Ebene geclippt [34]).

B.1.4 Darstellen der Flächen im Fill-Modus

Falls der Wireframe-Modus deaktiviert ist, müssen die Flächen des Tesserakts oder Pentachorons ausgefüllt dargestellt werden. Dieser Code ist noch aufwendiger als der vorige, deshalb folgt hier eine kurze Erklärung. Es wird wieder die Konfiguration des passenden Modells geladen (man geht auch je nach Modell in den passenden Polygonmodus), und bei Parallelprojektion einfach die w-Koordinate verworfen.

Die korrekte Perspektivprojektion des Pentachorons wurde noch nicht implementiert. Aber den Vorgang kann man aus den folgenden Zeilen erkennen: es wird für jede Ecke überprüft, ob sie erstens $w > 0$ hat, was bedeutet, dass sie normal gezeichnet wird. Falls nicht, wird für alle benachbarten Ecken überprüft, ob eine davon $w > 0$ hat, was dazu führt, dass die aktuelle Ecke zu dieser hin projiziert wird.

Das visuelle Ergebnis ist eher gewöhnungsbedürftig. Es wird empfohlen, die Objekte im Wireframe-Modus zu betrachten.

```
if(h.m==0) glBegin(GL_QUADS);
else if(h.m==1) glBegin(GL_TRIANGLES);
for(i=0; i<h.numF; i++) {
    if(h.m==0) {
        x1=h.v[h.cubeF[i][0]][X]; y1=h.v[h.cubeF[i][0]][Y];
            z1=h.v[h.cubeF[i][0]][Z]; w1=h.v[h.cubeF[i][0]][W];
        x2=h.v[h.cubeF[i][1]][X]; y2=h.v[h.cubeF[i][1]][Y];
            z2=h.v[h.cubeF[i][1]][Z]; w2=h.v[h.cubeF[i][1]][W];
        x3=h.v[h.cubeF[i][2]][X]; y3=h.v[h.cubeF[i][2]][Y];
            z3=h.v[h.cubeF[i][2]][Z]; w3=h.v[h.cubeF[i][2]][W];
        x4=h.v[h.cubeF[i][3]][X]; y4=h.v[h.cubeF[i][3]][Y];
            z4=h.v[h.cubeF[i][3]][Z]; w4=h.v[h.cubeF[i][3]][W];
    }
    else if(h.m==1) {
        w1=h.v[h.simplexF[i][0]][W];
        w2=h.v[h.simplexF[i][1]][W];
        w3=h.v[h.simplexF[i][2]][W];
    }
    if(h.proj==0) {
        if(h.m==0) {
```

```
            glVertex3fv(h.v[h.cubeF[i][0]]);
            glVertex3fv(h.v[h.cubeF[i][1]]);
            glVertex3fv(h.v[h.cubeF[i][2]]);
            glVertex3fv(h.v[h.cubeF[i][3]]);
        } else if(h.m==1) {
            glVertex3fv(h.v[h.simplexF[i][0]]);
            glVertex3fv(h.v[h.simplexF[i][1]]);
            glVertex3fv(h.v[h.simplexF[i][2]]);
        }
    } else {
        if(h.m==0) {
            if(w1>0.001) {
                glVertex4f(x1,y1,z1,w1);
            } else {
                if(w2>0.001) {
                    mu=(0.001-w2)/(w1-w2);
                    x5=(1-mu)*x2+mu*x1; y5=(1-mu)*y2+mu*y1; z5=(1-mu)*z2+mu*z1;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                } else if(w4>0.001) {
                    mu=(0.001-w4)/(w1-w4);
                    x5=(1-mu)*x4+mu*x1; y5=(1-mu)*y4+mu*y1; z5=(1-mu)*z4+mu*z1;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                }
            }
            if(w2>0.001) {
                glVertex4f(x2,y2,z2,w2);
            } else {
                if(w3>0.001) {
                    mu=(0.001-w3)/(w2-w3);
                    x5=(1-mu)*x3+mu*x2; y5=(1-mu)*y3+mu*y2; z5=(1-mu)*z3+mu*z2;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                } else if(w1>0.001) {
                    mu=(0.001-w1)/(w2-w1);
                    x5=(1-mu)*x1+mu*x2; y5=(1-mu)*y1+mu*y2; z5=(1-mu)*z1+mu*z2;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                }
            }
            if(w3>0.001) {
                glVertex4f(x3,y3,z3,w3);
            } else {
                if(w2>0.001) {
                    mu=(0.001-w2)/(w3-w2);
                    x5=(1-mu)*x2+mu*x3; y5=(1-mu)*y2+mu*y3; z5=(1-mu)*z2+mu*z3;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                } else if(w4>0.001) {
```

```
                    mu=(0.001-w4)/(w3-w4);
                    x5=(1-mu)*x4+mu*x3; y5=(1-mu)*y4+mu*y3; z5=(1-mu)*z4+mu*z3;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                }
            }
            if(w4>0.001) {
                glVertex4f(x4,y4,z4,w4);
            } else {
                if(w3>0.001) {
                    mu=(0.001-w3)/(w4-w3);
                    x5=(1-mu)*x3+mu*x4; y5=(1-mu)*y3+mu*y4; z5=(1-mu)*z3+mu*z4;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                } else if(w1>0.001) {
                    mu=(0.001-w1)/(w4-w1);
                    x5=(1-mu)*x1+mu*x4; y5=(1-mu)*y1+mu*y4; z5=(1-mu)*z1+mu*z4;
                        w5=0.001;
                    glVertex4f(x5,y5,z5,w5);
                }
            }
        } else if(h.m==1) {
         /* provisorische Lösung: */
            glVertex4fv(h.v[h.simplexF[i][0]]);
            glVertex4fv(h.v[h.simplexF[i][1]]);
            glVertex4fv(h.v[h.simplexF[i][2]]);
        }
    }
}
```

B.1.5 Umschalten der Modelle

Man darf nicht vergessen, beim Wechseln der Modelle auch die Anzahl der zugehörigen Ecken, Kanten und Flächen anzupassen. Sonst werden beide Modelle fehlerhaft dargestellt. Dazu gibt es folgenden Code, der den entsprechenden ASCII-Code oder den entsprechenden Menüeintrag abfängt:

```
h.m=!h.m;
if(h.m==0) {
    h.numV=16; h.numE=32; h.numF=24;
} else if(h.m==1) {
    h.numV=5; h.numE=10; h.numF=10;
}
```

Hier schaltet man zunächst das Flag `h.m` um und korrigiert dann die drei Grenzvariablen.

...

Anhang C

Nutzung des Demoprogramms

Das Demoprogramm ist als Quellcode oder als fertige unter Linux ausführbare Datei erhältlich.

C.1 Kompilierung

Das Demoprogramm `Cube` besteht aus der einzelnen Datei `cube.c`, die mithilfe der mitgelieferten Header sowie der selbst zusammengestellten Libraries übersetzt werden kann.

Prozedur 1: Verwendung der Makefile

Die mitgelieferte `Makefile` kann verwendet werden. Dazu muss man noch im Verzeichnis `./libs` die für sich passenden Libraries einfügen.

LIB	Windows x86	Linux
OpenGL	`OPENGL32.LIB`	`libGL.so`
GLU	`GLU32.LIB`	`libGLU.so`
GLUT	`glut32.lib`	`libglut.so`

Tabelle C.1: Bibliotheksbezeichnungen

Die Headerdateien (`GL.H`, `GLU.H` und `glut.h`) liegen bereits in `./libs/GL` vor. Schließlich führt die Makefile folgenden Befehl aus:

```
gcc -I./libs -L./libs -o cube cube.c -lGL -GLU -lglut -lm
```

Prozedur 2: Verwendung von Microsoft Visual Studio

Wie in Tabelle C.1 angegeben, sollten noch die passenden Bibliotheken in das Verzeichnis `./libs` gelegt werden.

Dann importiere man `cube.c` in Visual Studio, und nach einem Verweis auf den Header- und Bibliotheksordner `./libs` sollte sich das Programm einfach bauen und ausführen lassen.

C.2 Verwendung

Das Programm lässt sich entweder mit der Tastatur oder mit der rechten Maustaste bedienen.

Globale Optionen

Die globalen Optionen verändern das System an sich und können immer aktiviert werden.

Funktion	Tastatureingabe	Menüeintrag
Stereo/Mono	`M`	`Stereo/Mono`
Modell wechseln	`1 / 2 / 3 / 4`	`Switch Model`
Licht an/aus	`L`	`System - Lighting on/off`
Depth Test an/aus	`D`	`System - Depth test on/off`
Culling an/aus	`C`	`System - Cull face on/off`
Fill/Wireframe	`W`	`System - Wireframe on/off`
Achsen an/aus	`-`	`System - Axes on/off`

Spezifische Optionen

Für Modell `1` (Würfel) gibt es folgende spezielle Optionen:

Funktion	Tastatureingabe	Menüeintrag
Rotation in x-Richtung	`a / d`	-
Rotation in y-Richtung	`w / s`	-
Rotation in z-Richtung	`q / e`	-
Transformation in $\pm z$-Richtung	`r / f`	-
Animation (Auto-Rotation)	`G`	`Cube - Rotate on/off`

Modell `2` (Kugelsystem):

Funktion	Tastatureingabe	Menüeintrag
Animation (Beschleunigung)	`G`	`Ball - Animation on/off`

Modell `3` (Bézierfläche):

Funktion	Tastatureingabe	Menüeintrag
Rotation in x-Richtung	`a / d`	-
Rotation in y-Richtung	`w / s`	-
Rotation in z-Richtung	`q / e`	-
Kontrollpunkte an/aus	`P`	`Curve - Control points on/off`

Modell `4` ('Hyperspace'):

Funktion	Tastatureingabe	Menüeintrag
Rotation entlang xy-Ebene	`q`	-
Rotation entlang yz-Ebene	`w`	-
Rotation entlang xz-Ebene	`e`	-
Rotation entlang xw-Ebene	`a`	-
Rotation entlang yw-Ebene	`s`	-
Rotation entlang zw-Ebene	`d`	-
Parallel-/Perspektivprojektion	`P`	`Hyperspace - Projection par./persp.`
Tesserakt/Pentachoron	`H`	`Hyperspace - Change 4D model`

Anhang D

Stereobilder

In diesem Kapitel sind weitere interessante Stereogramme und Anaglyphen zusammengefasst.

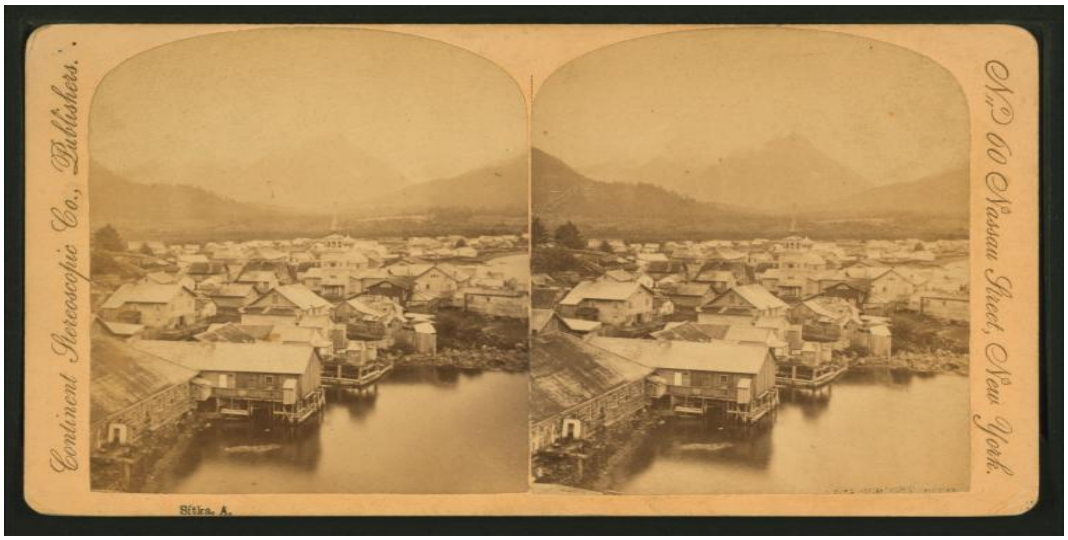


Abbildung D.1: Sitka, Alaska (Stereogramm) [26]

Abbildung D.2: Am Allenkaket River, Alaska (Stereogramm) [27]

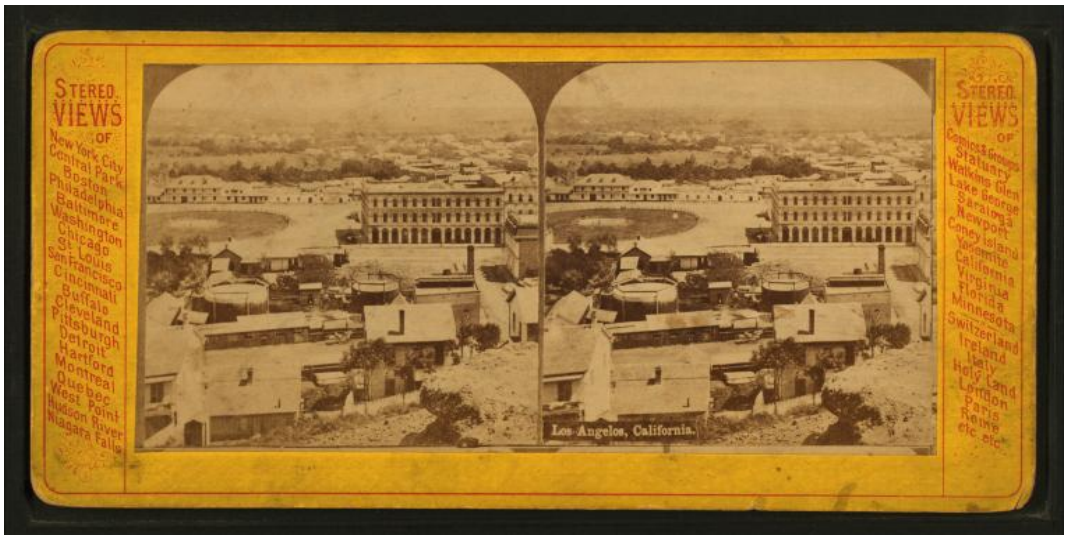


Abbildung D.3: Los Angeles, California (Stereogramm) [28]

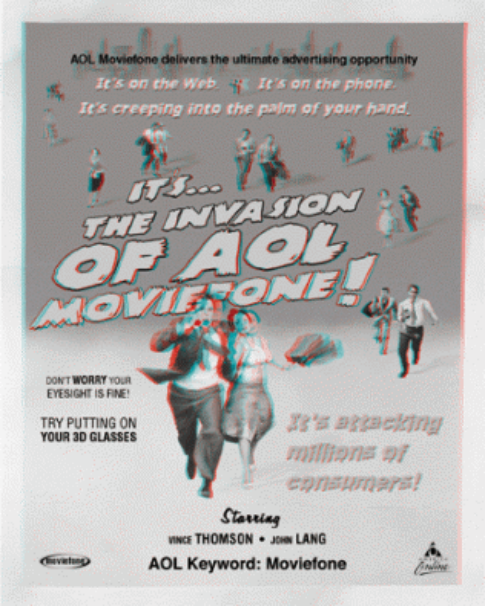


Abbildung D.4: AOL-Moviefone-Werbeplakat (Anaglyph) [29]

Abbildung D.5: New York City, New York (Anaglyph) [29]

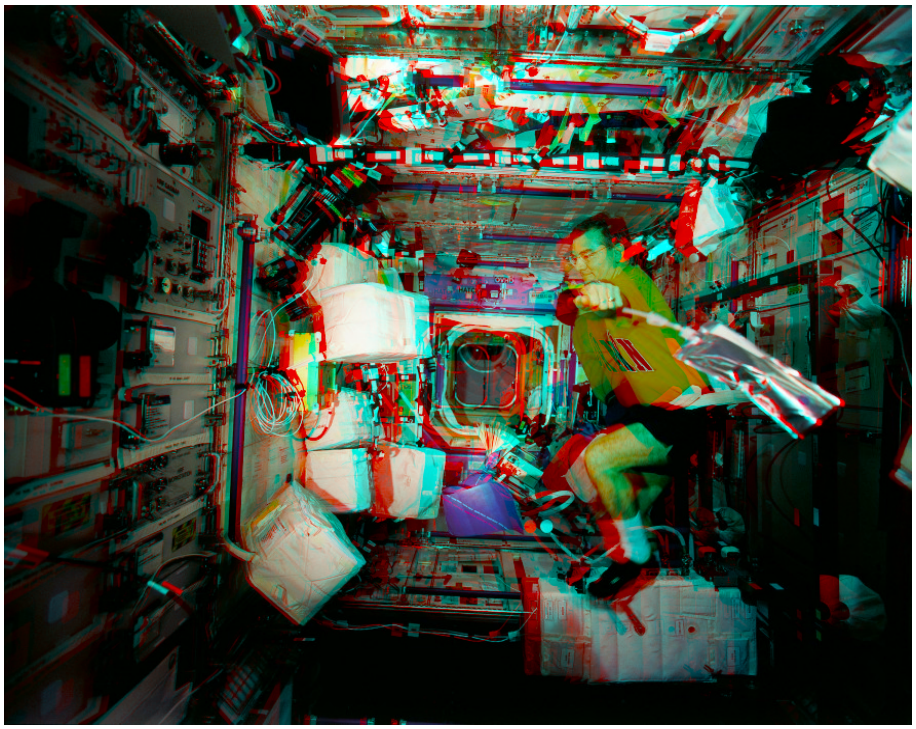


Abbildung D.6: In der International Space Station (Anaglyph) [30]

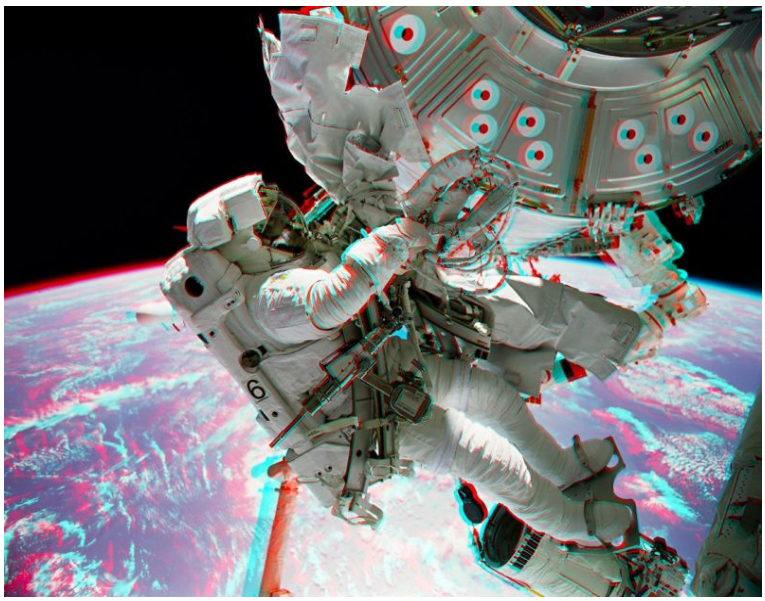


Abbildung D.7: Astronaut außerhalb der International Space Station (Anaglyph) [30]

Literaturverzeichnis

[1] Henry George Liddell, Robert Scott. A Greek-English Lexicon.
http://perseus.tufts.edu/hopper

[2] Samuel Gateau, Steve Nash. Implementing Stereoscopic 3D in Your Applications.
NVIDIA GPU Technology Conference, 2010.
http://nvidia.com/content/GTC-2010/pdfs/2010_GTC2010.pdf

[3] R. Blake, R. Sekuler. Perception, 5th edition.
McGraw-Hill, 2005, New York.

[4] Aristovoulos Christidis. bi_objGL.c. Mai 2017.
https://homepages.thm.de/~hg11237/Start/01Lehre/10SwPBG/SwPBGpix&refs/SwPBGsw.zip

[5] Animesh Mishra. Rendering 3D Anaglyph in OpenGL. Mai 2011.
http://www.animesh.me/2011/05/rendering-3d-anaglyph-in-opengl.html

[6] Robert S. Allison. The Camera Convergence Problem Revisited.
http://percept.eecs.yorku.ca/papers/Allison-%20Camera%20convergence%20problem%20revisited.pdf

[7] Paul Bourke. Creating and Viewing Anaglyphs. Juni 2000.
http://paulbourke.net/stereographics/anaglyph

[8] GL.H, v1.2 2003/02/09
Mesa 3-D graphics library, version 4.0
Copyright 1999-2001 Brian Paul.

[9] glut.h
GLUT API, version 3
Copyright 1994, 1995, 1996, 1998 Mark J. Kilgard.

[10] OpenGL Programming Guide - Chapter 5. Lighting.
http://glprogramming.com/red/chapter05.html

[11] OpenGL Programming Guide - Chapter 3. Viewing.
http://glprogramming.com/red/chapter03.html

[12] OpenGL Programming Guide - Chapter 10. The Framebuffer.
http://glprogramming.com/red/chapter10.html

[13] OpenGL Programming Guide - Chapter 4. Color.
http://glprogramming.com/red/chapter04.html

[14] Eric Weisstein. Matrix Multiplication.
MathWorld – A Wolfram Web Resource.
http://mathworld.wolfram.com/MatrixMultiplication.html

[15] Daniel Wexler. Stereo Rendering. Juni 2006.
https://www.nvidia.com/object/page_pulled_off_gz_stereo.html

[16] Mark Kilgard. Window Management.
The OpenGL Utility Toolkit (GLUT) Programming Interface API, Version 3. Silicon Graphics, Inc. 1994, 1995, 1996.
https://www.opengl.org/resources/libraries/glut/spec3/node15.html

[17] Mark Kilgard. Beginning Event Processing.
The OpenGL Utility Toolkit (GLUT) Programming Interface API, Version 3. Silicon Graphics, Inc. 1994, 1995, 1996.
https://www.opengl.org/resources/libraries/glut/spec3/node13.html

[18] Adaptive VSync.
https://www.geforce.com/hardware/technology/adaptive-vsync/technology

[19] Jim Lucas. What Is Parallax? August 2005.
https://space.com/30417-parallax.html

[20] Mark Kilgard. Callback Registration.
The OpenGL Utility Toolkit (GLUT) Programming Interface API, Version 3. Silicon Graphics, Inc. 1994, 1995, 1996.
https://www.opengl.org/resources/libraries/glut/spec3/node45.html

[21] Chapter 15, Collision Theory.
web.mit.edu/8.01t/www/materials/modules/chapter15.pdf

[22] View Frustum Culling.
http://www.lighthouse3d.com/tutorials/view-frustum-culling

[23] Harold A. Layer. Stereoscopy: Where Did It Come From? Where Will It Lead?
http://userwww.sfsu.edu/hl/stereo.html

[24] NYPL Labs. Stereoscopic photography.
The New York Public Library. 2011-2018.
http://stereo.nypl.org/about/stereoscopy

[25] B. W. Kilburn. On the way to Klondyke. 1898.
Robert N. Dennis collection of stereoscopic views. NYPL Digital Collections.
https://digitalcollections.nypl.org/items/510d47e0-2c11-a3d9-e040-e00a18064a99

[26] Sitka, Alaska.
Robert N. Dennis collection of stereoscopic views. NYPL Digital Collections.
https://digitalcollections.nypl.org/items/510d47e0-2bbd-a3d9-e040-e00a18064a99

[27] B. L. Singley. A letter from Home – on the Allenkaket River, Alaska.
Robert N. Dennis collection of stereoscopic views. NYPL Digital Collections.
https://digitalcollections.nypl.org/items/510d47e0-2baf-a3d9-e040-e00a18064a99

[28] Los Angeles, California.
Robert N. Dennis collection of stereoscopic views. NYPL Digital Collections.
https://digitalcollections.nypl.org/items/510d47e0-3221-a3d9-e040-e00a18064a99

[29] 3D Anaglyph Image Gallery.
American Paper Optics, 2010.
http://www.3dglassesonline.com/our-products/technologies/anaglyph-image-gallery

[30] Eric Dubois. red-cyan anaglyphs.
https://www.flickr.com/photos/e_dubois/albums/72157606640245479

[31] OpenGL Programming Guide - Chapter 12. Evaluators and NURBS.
http://glprogramming.com/red/chapter12.html

[32] Paul Bourke. Bézier curves. April 1989, Dezember 1996.
http://paulbourke.net/geometry/bezier

[33] Paul Bourke. Regular Polytopes (Platonic solids) in 4D. Juni 1997, November 2003.
http://paulbourke.net/geometry/hyperspace

[34] Steven Richard Hollasch. Four-Space Visualization of 4D Objects.
Master Thesis, Arizona State University, August 1991.
https://hollasch.github.io/ray4/Four-Space_Visualization_of_4D_Objects.html

[35] 4D Visualization: Projections.
http://www.eusebeia.dyndns.org/4d/vis/05-proj-1

[36] OpenGL Programming Guide - Appendix F. Homogenous Coordinates and Transformation Matrices.
http://glprogramming.com/red/appendixf.html

[37] Glenn Research Center. The Drag Equation.
https://www.grc.nasa.gov/WWW/K-12/airplane/drageq.html

Alle Quellen: Stand 9. April 2018