Kapitel 21

Ray Tracing

21.1 Grundlagen

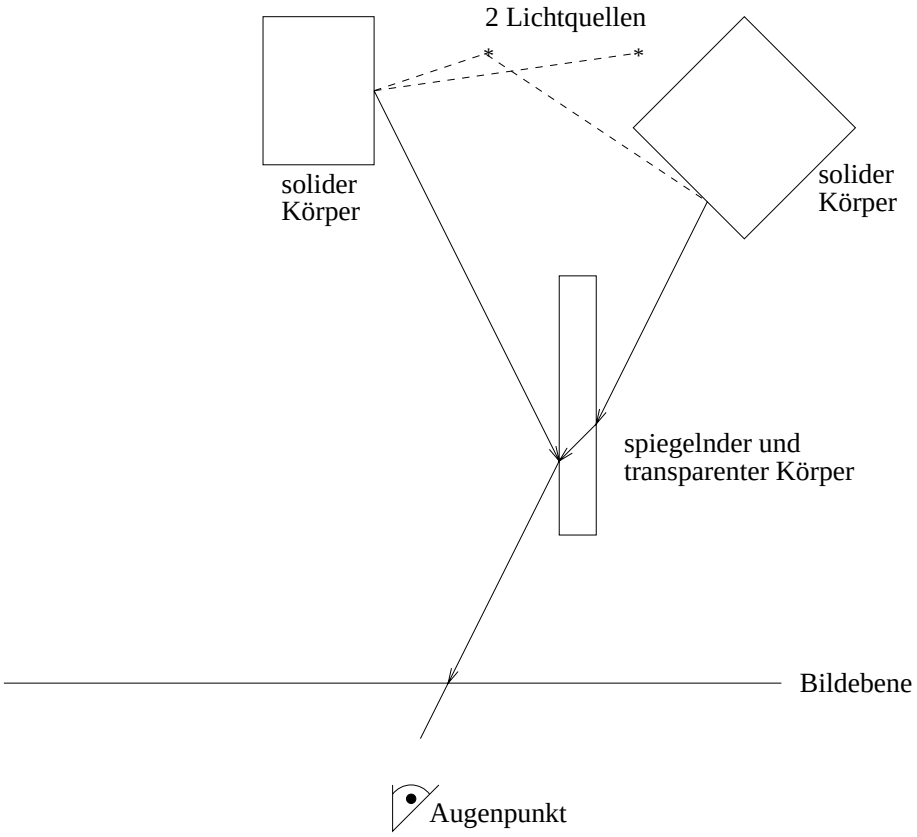


Abbildung 21.1: Prinzip der Strahlverfolgung

Verfahren mit *Ray Tracing* (Strahlverfolgung) eignen sich ausgezeichnet zur Modellierung von spiegelnden Reflexionen und von Transparenz mit Brechung (ohne Streuung). Globale Lichter werden mit einem ambienten Beleuchtungsterm ohne Richtung behandelt. Ausgehend

vom Augenpunkt wird durch jedes Pixel ein Strahl gelegt und der Schnittpunkt dieses Strahls mit dem ersten getroffenen Objekt bestimmt. Trifft der Strahl auf kein Objekt, so erhält das Pixel die Hintergrundfarbe. Ist das Objekt spiegelnd, so wird der Reflexionsstrahl berechnet und rekursiv weiterbehandelt. Ist das Objekt transparent, wird zusätzlich der gebrochene Strahl weiterbehandelt. Zur Berechnung von Schatten wird von jedem Schnittpunkt zwischen Strahl und Objekt zu jeder Lichtquelle ein zusätzlicher Strahl ausgesandt. Trifft dieser Strahl auf ein blockierendes Objekt, dann liegt der Schnittpunkt im Schatten dieser Lichtquelle, und das von ihr ausgestrahlte Licht geht in die Intensitätsberechnung des Punktes nicht ein.

21.2 Ermittlung sichtbarer Flächen durch Ray Tracing

Verfahren mit *Ray Tracing* ermitteln die Sichtbarkeit von Flächen, indem sie imaginäre Lichtstrahlen des Betrachters zu den Objekten der Szene verfolgen. Man wählt ein Projektionszentrum (das Auge des Betrachters) und ein Window in einer beliebigen Bildebene. Das Window wird durch ein regelmäßiges Gitter aufgeteilt, dessen Elemente den Pixeln in der gewünschten Auflösung entsprechen. Dann schickt man für jedes Pixel im Window einen *Augstrahl* vom Projektionszentrum durch den Mittelpunkt des Pixels auf die Szene. Das Pixel erhält die Farbe des ersten getroffenen Objekts. Das folgende Programm enthält den Pseudocode für diesen einfachen Ray Tracer.

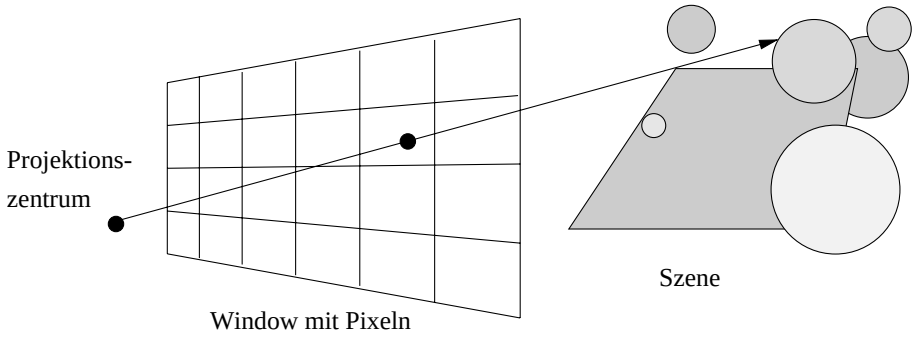


Abbildung 21.2: Strahl durch Bildebene

```
Wähle Projektionszentrum und Window in der Bildebene;
for(jede Rasterzeile des Bildes){
   for(jedes Pixel der Rasterzeile){
     Berechne den Strahl vom Projektionszentrum durch das Pixel;
     for(jedes Objekt der Szene){
        if(Objekt wird geschnitten und liegt bisher am nächsten)
           Speichere Schnittpunkt und Name des Objekts;
     }
     Setze das Pixel auf die Farbe des nächstliegenden Objektschnittpunkts;
   }
}
```

21.3 Berechnung von Schnittpunkten

Hauptaufgabe eines jeden Ray Tracers ist es, den Schnittpunkt eines Strahls mit einem Objekt zu bestimmen. Man benutzt dazu die parametrisierte Darstellung eines Vektors. Jeder Punkt (x, y, z) auf dem Strahl von (x_0, y_0, z_0) nach (x_1, y_1, z_1) wird durch einen bestimmten Wert definiert mit

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

Zur Abkürzung definiert man $\Delta x, \Delta y$ und Δz als

$$\Delta x = x_1 - x_0, \quad \Delta y = y_1 - y_0, \quad \Delta z = z_1 - z_0 \ .$$

Damit kann man schreiben

$$x = x_0 + t\Delta x, \quad y = y_0 + t\Delta y, \quad z = z_0 + t\Delta z \ .$$

Ist (x_0, y_0, z_0) das Projektionszentrum und (x_1, y_1, z_1) der Mittelpunkt eines Pixels im Window, so durchläuft t zwischen diesen Punkten die Werte von Null bis Eins. Negative Werte für t liefern Punkte hinter dem Projektionszentrum, Werte größer als Eins stellen Punkte auf der Seite des Windows dar, die vom Projektionszentrum abgewandt ist. Man braucht für jeden Objekttyp eine Darstellung, mit der man den Wert von t am Schnittpunkt des Strahls mit dem Objekt bestimmen kann. Die Kugel bietet sich dafür als eines der einfachsten Objekte an. Aus diesem Grund tauchen Kugeln auch so oft in Ray-Tracing-Bildern auf. Eine Kugel um den Mittelpunkt (a, b, c) und Radius r kann durch die Gleichung

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2$$

dargestellt werden. Zur Berechnung des Schnittpunkts multipliziert man die Gleichung aus und setzt x, y und z aus der Gleichung von oben ein.

Man erhält eine quadratische Gleichung in t, deren Koeffizienten nur die Konstanten aus den Gleichungen der Kugel und des Strahls enthalten. Man kann sie also mit der Lösungsformel für quadratische Gleichungen lösen. Wenn es keine reellen Lösungen gibt, schneidet der Strahl die Kugel nicht. Gibt es genau eine Lösung, dann berührt der Strahl die Kugel. Andernfalls geben die beiden Lösungen die Schnittpunkte mit der Kugel an. Der Schnittpunkt mit dem kleinsten positiven t-Wert liegt am nächsten.

Man muß zur Schattierung der Fläche die Flächennormale im Schnittpunkt berechnen. Im Fall der Kugel ist das besonders einfach, weil die (nicht normalisierte) Normale einfach der Vektor vom Kugelmittelpunkt zum Schnittpunkt ist: Die Kugel mit Mittelpunkt (a, b, c) hat im Schnittpunkt (x, y, z) die Flächennormale $((x - a)/r, (y - b)/r, (z - c)/r)$.

Es ist etwas schwieriger, den Schnittpunkt des Strahls mit einem Polygon zu berechnen. Um festzustellen, ob der Strahl ein Polygon schneidet, testet man zuerst, ob der Strahl die Ebene des Polygons schneidet und anschließend, ob der Schnittpunkt innerhalb des Polygons liegt.

21.4 Effizienzsteigerung zur Ermittlung sichtbarer Flächen

Die vorgestellte einfache, aber rechenintensive Version des Ray-Tracing-Algorithmus schneidet jeden Augstrahl mit jedem Objekt der Szene. Für ein Bild der Größe 1024×1024 mit 100 Ob-

jekten wären daher 100 Mio. Schnittpunktberechnungen erforderlich. Ein System verbraucht bei typischen Szenen 75-95 Prozent der Rechenzeit für die Schnittpunktroutine. Daher konzentrieren sich die Ansätze zur Effizienzsteigerung auf die Beschleunigung der Schnittpunktberechnungen oder deren völlige Vermeidung.

Optimierung der Schnittpunktberechnungen

Die Formel für den Schnittpunkt mit einer Kugel läßt sich verbessern. Wenn man die Strahlen so transformiert, daß sie entlang der z-Achse verlaufen, kann man die gleiche Transformation auf die getesteten Objekte anwenden. Dann liegen alle Schnittpunkte bei $x = y = 0$. Dieser Schritt vereinfacht die Berechnung der Schnittpunkte. Das nächstliegende Objekt erhält man durch Sortieren der z-Werte. Mit der inversen Transformation kann man den Schnittpunkt dann für die Schattierungsberechnungen zurücktransformieren.

Begrenzungsvolumina eignen sich besonders gut dazu, die Zeit für die Berechnung der Schnittpunkte zu reduzieren. Man umgibt ein Objekt, für das die Schnittpunktberechnungen sehr aufwendig sind, mit einem Begrenzungsvolumen, dessen Schnittpunkte einfacher zu berechnen sind, z.B. Kugel, Ellipsoid oder Quader. Das Objekt wird nur dann getestet, wenn der Strahl das Begrenzungsvolumen schneidet.

Hierarchien

Begrenzungsvolumina legen zwar selbst noch keine Reihenfolge oder Häufigkeit der Schnittpunkttests fest. Sie können aber in verschachtelten Hierarchien organisiert werden. Dabei bilden die Objekte die Blätter, die inneren Knoten begrenzen ihre Söhne. Hat ein Strahl keinen Schnittpunkt mit einem Vaterknoten, dann gibt es garantiert auch keinen Schnittpunkt mit einem seiner Söhne. Beginnt der Schnittpunkttest also an der Wurzel, dann entfallen ggf. trivialerweise viele Zweige der Hierarchie (und damit viele Objekte).

Bereichsunterteilung

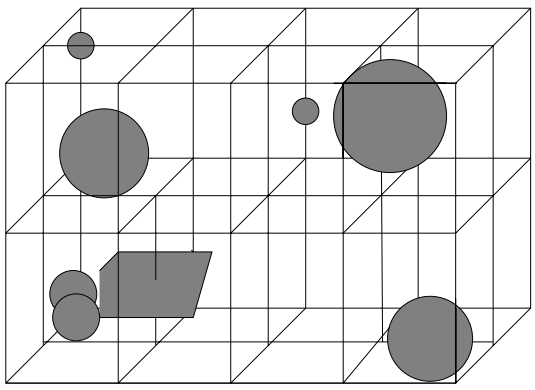


Abbildung 21.3: Unterteilung der Szene durch ein regelmäßiges Gitter

Die Hierarchie von Begrenzungsvolumina organisiert die Objekte von unten nach oben. Die Bereichsunterteilung teilt dagegen von oben nach unten auf. Zuerst berechnet man die *bounding box* der ganzen Szene. Bei einer Variante unterteilt man die *bounding box* dann in ein regelmäßiges Gitter gleich großer Bereiche. Jedem Bereich wird eine Liste mit den Objekten zugewiesen, die entweder ganz oder teilweise darin enthalten sind. Zur Erzeugung der Listen weist man jedes Objekt einem oder mehreren Bereichen zu, die dieses Objekt enthalten. Ein

Strahl muß jetzt nur noch mit den Objekten in den durchlaufenen Bereichen geschnitten werden. Außerdem kann man die Bereiche in der Reihenfolge untersuchen, in der sie vom Strahl durchlaufen werden. Sobald es in einem Bereich einen Schnittpunkt gibt, muß man keine weiteren Bereiche mehr testen. Man muß jedoch alle anderen Objekte des Bereichs untersuchen, um das Objekt mit dem nächstliegenden Schnittpunkt zu ermitteln.

21.5 Rekursives Ray Tracing

In diesem Abschnitt wird der Basisalgorithmus zum Ray Tracing auf die Behandlung von Schatten, Reflexion und Brechung erweitert. Der einfache Algorithmus ermittelte die Farbe eines Pixels am nächsten Schnittpunkt eines Augstrahls und eines Objekts mit einem beliebigen der früher beschriebenen Beleuchtungsmodelle. Zur Schattenberechnung wird ein zusätzlichen Strahl vom Schnittpunkt zu allen Lichtquellen geschickt. Schneidet einer dieser *Schattenstrahlen* auf seinem Weg ein Objekt, dann liegt das Objekt am betrachteten Punkt im Schatten, und der Schattierungsalgorithmus ignoriert den Beitrag der Lichtquelle dieses Schattenstrahls.

Der rekursive Ray-Tracing-Algorithmus startet zusätzlich zu den Schattenstrahlen weitere *Spiegelungs-* und *Brechungsstrahlen* im Schnittpunkt (siehe Abbildung 21.4). Diese Spiegelungs-, Reflexions- und Brechungsstrahlen heißen *Sekundärstrahlen*, um sie von den *Primärstrahlen* zu unterscheiden, die vom Auge ausgehen. Gibt es bei dem Objekt spiegelnde Reflexion, dann wird ein Spiegelungsstrahl um die Flächennormale in Richtung r gespiegelt. Ist das Objekt transparent, und es gibt keine totale innere Reflexion, startet man einen Brechungsstrahl entlang t in das Objekt. Der Winkel wird nach dem Gesetz von Snellius bestimmt.

Jeder dieser Reflexions- und Brechungsstrahlen kann wiederum neue Spiegelungs-, Reflexions- und Brechungsstrahlen starten.

Typische Bestandteile eines *Ray-Tracing*-Programms sind die Prozeduren `shade` und `intersect`, die sich gegenseitig aufrufen. `shade` berechnet die Farbe eines Punktes auf der Oberfläche. Hierzu werden die Beiträge der reflektierten und gebrochenen Strahlen benötigt. Diese Beiträge erfordern die Bestimmung der nächstgelegenen Schnittpunkte, welche die Prozedur `intersect` liefert. Der Abbruch der Strahlverfolgung erfolgt bei Erreichen einer vorgegebenen Rekursionstiefe oder wenn ein Strahl kein Objekt trifft.

Seien v, N die normierten Vektoren für Sehstrahl und Normale. Sei $n = n_2/n_1$ das Verhältnis der beiden Brechungsindizes. Dann sind die Vektoren r und t, die im Schnittpunkt P zur Weiterverfolgung des reflektierten und gebrochenen Strahls benötigt werden, definiert durch

$$\begin{aligned} r &= v + 2\cos(\phi)N \\ t &= (-\cos(\phi) - q)N + v \\ \text{mit } q &= \sqrt{n^2 - 1 + \cos^2(\phi)} \end{aligned}$$

Ist p ein imaginärer Ausdruck, liegt Totalreflexion vor, und der Transmissionsanteil wird gleich Null gesetzt.

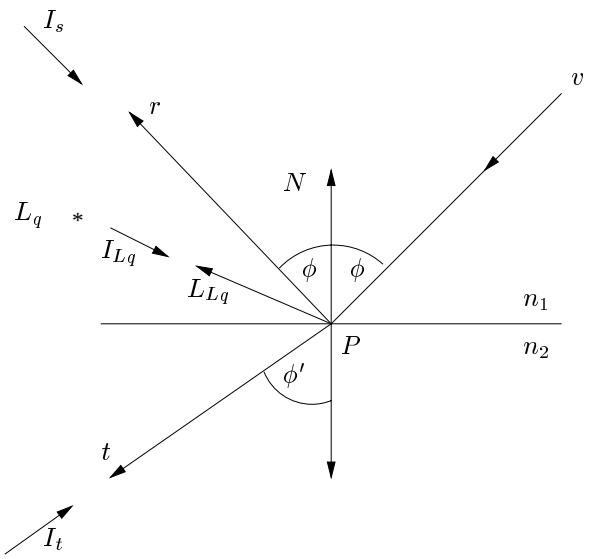


Abbildung 21.4: Einflußgrößen für Punkt P

In die Berechnung der Farbe im Punkt P geht die gewichtete Summe der in den einzelnen Rekursionsschritten berechneten Farbintensitäten ein. Unter Berücksichtigung des *Phong*-Modells ergibt sich

$$I = k_a I_H + k_d \sum_{Lq} I_{Lq}(N \cdot L_{Lq}) + k_s(I_s + \sum_{Lq} I_{Lq}(A \cdot R_{Lq})^c) + k_t I_t$$

mit

k_a	ambienter Reflexionskoeffizient
I_H	Hintergrundbeleuchtung
k_d	diffuser Reflexionskoeffizient
I_{Lq}	Intensität der Lichtquelle
N	Normale auf Ebene
L_{Lq}	Vektor zur Lichtquelle
k_s	spiegelnder Reflexionskoeffizient
I_s	globale Reflexionssintensität
A	Vektor zum Augenpunkt
R_{Lq}	Reflexionsvektor für Lichtquelle L_q
c	spekularer Exponent
k_t	Transmissionskoeffizient (materialabhängig)
I_t	Transmissionsintensität

Pseudocode für einfaches rekursives Ray Tracing

```
Wähle Projektionszentrum und Window in der view plane;
for(jede Rasterzeile des Bildes){
   for(jedes Pixel der Rasterzeile){
     Ermittle Strahl vom Projektionszentrum durch das Pixel;
     pixel = RT_intersect(ray, 1);
   }
}

/* Schneide den Strahl mit Objekten, und berechne die Schattierung am nächsten */
/* Schnittpunkt. Der Parameter depth ist die aktuelle Tiefe im Strahlenbaum. */

RT_color RT_intersect(RT_ray ray, int depth)
{
   Ermittle den nächstliegenden Schnittpunkt mit einem Objekt;
   if(Objekt getroffen){
     Berechne die Normale im Schnittpunkt;
     return RT_shade(nächstliegendes getroffenes Objekt, Strahl, Schnittpunkt,
                     Normale, depth);
   }
   else
     return HINTERGRUND_FARBE;
}



/* Berechne Schattierung für einen Punkt durch Verfolgen der Schatten-, */
/* Reflexions- und Brechungsstrahlen */

RT_color RT_shade(
   RT_object object,          /* Geschnittenes Objekt */
   RT_ray ray,                /* Einfallender Strahl */
   RT_point point,            /* Schnittpunkt, der schattiert werden soll */
   RT_normal normal,          /* Normale in dem Punkt */
   int depth)                 /* Tiefe im Strahlenbaum */
{
RT_color color;               /* Farbe des Strahls */
RT_ray rRay, tRay, sRay;      /* Reflexions-, Brechungs- und Schattenstrahlen */
RT_color rColor, tColor;      /* Farbe des reflektierten und gebrochenen Strahls */

   color = Term für das ambiente Licht;


   for(jede Lichtquelle){
     sRay = Strahl von der Lichtquelle zum Punkt;
     if(Skalarprodukt der Normalen mit der Richtung zum Licht ist positiv){
        Berechne, wieviel Licht von opaken und transparenten Flächen blockiert wird;
```

```
        Skaliere damit die Terme für diffuse und spiegelnde Reflexion, bevor sie
        zur Farbe addiert werden;
     }
}
if(depth < maxDepth){        /* Bei zu großer Tiefe beenden */
     if(Objekt reflektiert){
        rRay = Strahl vom Punkt in Reflexionsrichtung;
        rColor = RT_intersect(rRay, depth + 1);
        Skaliere rColor mit dem Spiegelungskoeffizienten,
        und addiere den Wert zur Farbe;
     }
     if(Objekt ist transparent){
        tRay = Strahl vom Punkt in Brechungsrichtung;
        if(es gibt keine totale interne Reflexion){
           tColor = RT_intersect(tRay, depth + 1);
           Skaliere tColor mit dem Transmissionskoeffizienten,
           und addiere den Wert zur Farbe;
        }
     }
   }
   return color;             /* Liefere die Farbe des Strahls zurück */
}
```

Ray Tracing ist besonders anfällig für Probleme, die durch die begrenzte Rechengenauigkeit entstehen. Dies zeigt sich vor allem bei der Berechnung der Schnittpunkte sekundärer Strahlen mit den Objekten. Wenn man die x-, y- und z-Koordinaten eines Objekts mit einem Augstrahl berechnet hat, dienen sie zur Definition des Startpunkts eines Sekundärstrahls. Für diesen muß dann der Parameter t bestimmt werden. Wird das eben geschnittene Objekt mit dem neuen Strahl geschnitten, hat t wegen der begrenzten Rechengenauigkeit oft einen kleinen Wert ungleich Null. Dieser falsche Schnittpunkt kann sichtbare Probleme bewirken.

21.6 Public Domain Ray Tracer Povray

Der "Persistence of Vision Ray Tracer" (POV-Ray) ist ein urheberrechtlich geschütztes Freeware-Programm zum Berechnen einer fotorealitischen Projektion aus einer 3-dimensionalen Szenenbeschreibung auf der Grundlage der Strahlverfolgung (`http://www.povray.org`). Seine Implementierung basiert auf DKBTrace 2.12 von David Buck & Aaron Collins. Zur Unterstützung des Anwenders gibt es einige include-Files mit vordefinierten Farben, Formen und Texturen.

```
#include "colors.inc"
#include "textures.inc"

camera {
       location <3, 3, -1>
       look_at  <0, 1,  2>
}
light_source { <0, 4, -3> color White}
light_source { <0, 6, -4> color White}
light_source { <6, 4, -3> color White}

plane {
       <0, 1, 0>, 0
       pigment {
               checker
                      color White
                      color Black
       }
}

sphere {
       <0, 2, 4>, 2
       texture {
               pigment {color Orange}
               normal  {bumps 0.4  scale 0.2}
               finish  {phong 1}
       }
}

box {
       <-1, 0, -0.5>,
       < 1, 2,  1.5>
       pigment {
               DMFWood4
               scale 2
       }
       rotate y*(-20)
}
```

Povray-Quelltext scene.pov

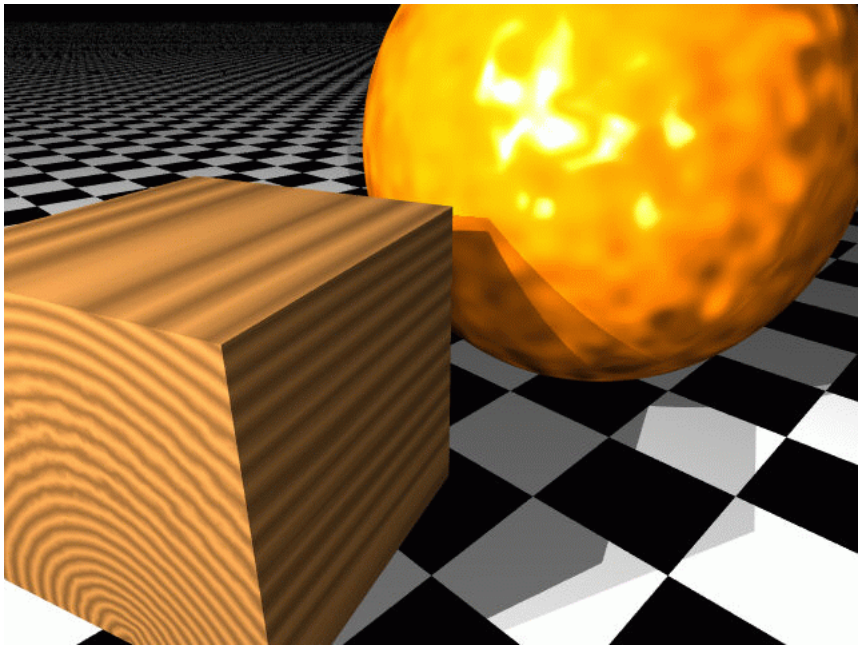


Abbildung 21.5: Povray-Bild zur Szenenbeschreibung

Der Aufruf von *povray* erfolgt zweckmäßigerweise mit einem Shell-Skript, dem der Name der zu bearbeitenden Datei als einziger Parameter übergeben wird:

```
#!/bin/sh

povray +I$1.pov +Oscene.tga +L/usr/lib/povray3/include +W320 +H200
```

Die vom Ray-Tracer erstellte Statistik lautet:

```
scene.pov Statistics, Resolution 320 x 200
-----------------------------------------------------------------------------
Pixels:           64000   Samples:          64000   Smpls/Pxl: 1.00
Rays:             64000   Saved:                0   Max Level: 1/5
-----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded  Percentage
-----------------------------------------------------------------------------
Box                             238037           40342     16.95
Plane                           238037           62400     26.21
Sphere                          238037           50948     21.40
-----------------------------------------------------------------------------
Calls to Noise:                      0   Calls to DNoise:          83403
-----------------------------------------------------------------------------
Shadow Ray Tests:           522111   Succeeded:                24497
-----------------------------------------------------------------------------
Smallest Alloc:                 12 bytes   Largest:            12308
Peak memory used:           103504 bytes
-----------------------------------------------------------------------------
Time For Trace:    0 hours  0 minutes  18.0 seconds (18 seconds)
    Total Time:    0 hours  0 minutes  18.0 seconds (18 seconds)
```

Kapitel 22

Caligary trueSpace

3D-Modellerierung unter Verwendung von

- hierarchischer Objektstruktur,
- Polygonvereinigung, -schnitt, -komplement,
- Translation, Skalierung, Rotation,
- Extrusion,
- Verformung,
- Drehkörpern.

Projektion und Rendern unter Berücksichtigung von

- Lichtquellen,
- Kamerastandpunkten,
- Spiegelung, Transparenz, Lichtbrechung,
- Texture Mapping, Bump Mapping.

Animation durch

- Keyframes,
- Pfade.

Ausgabe als

- Bitmapdateien,
- VRML-Welten,
- AVI-Files.

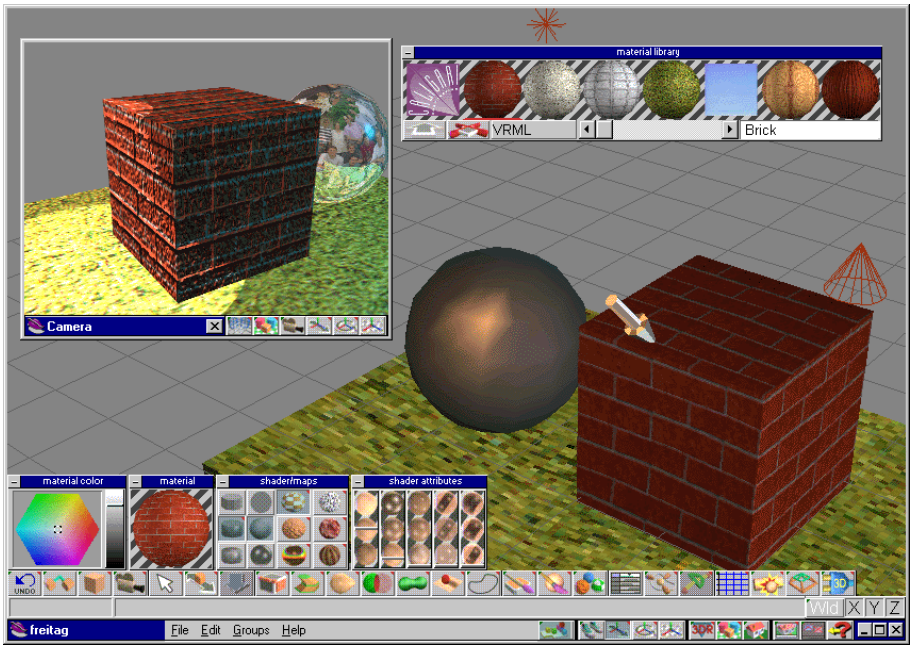


Abbildung 22.1: Screenshot vom 3D-Werkzeug Caligary trueSpace

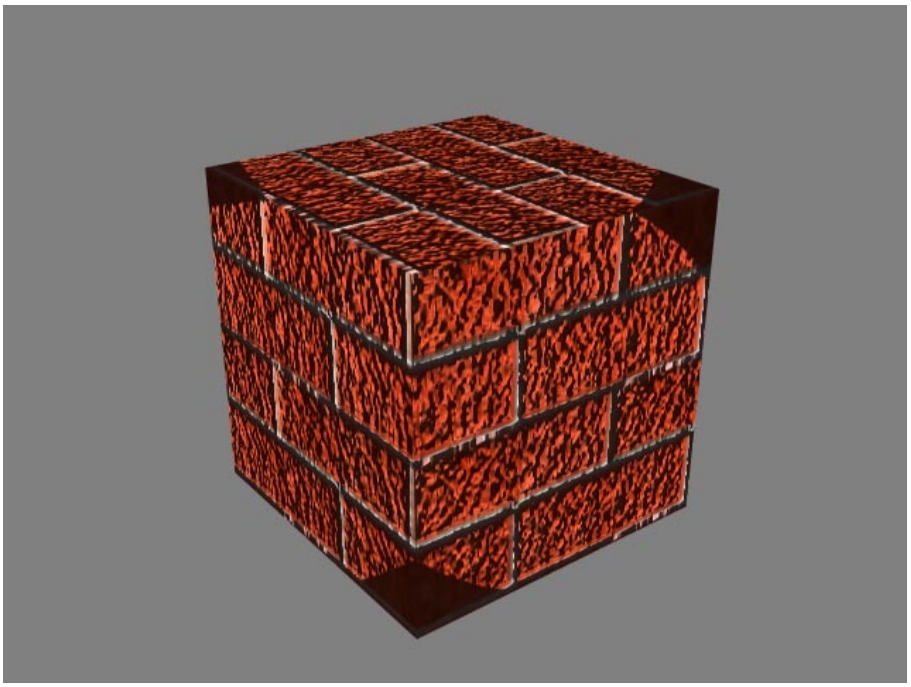


Abbildung 22.2: Würfel, gerendert mit Texture-Mapping, Bump-Mapping und Spotlicht

Abbildung 22.3: Kugel, gerendert mit spiegelnder Oberfläche und Environment-Textur

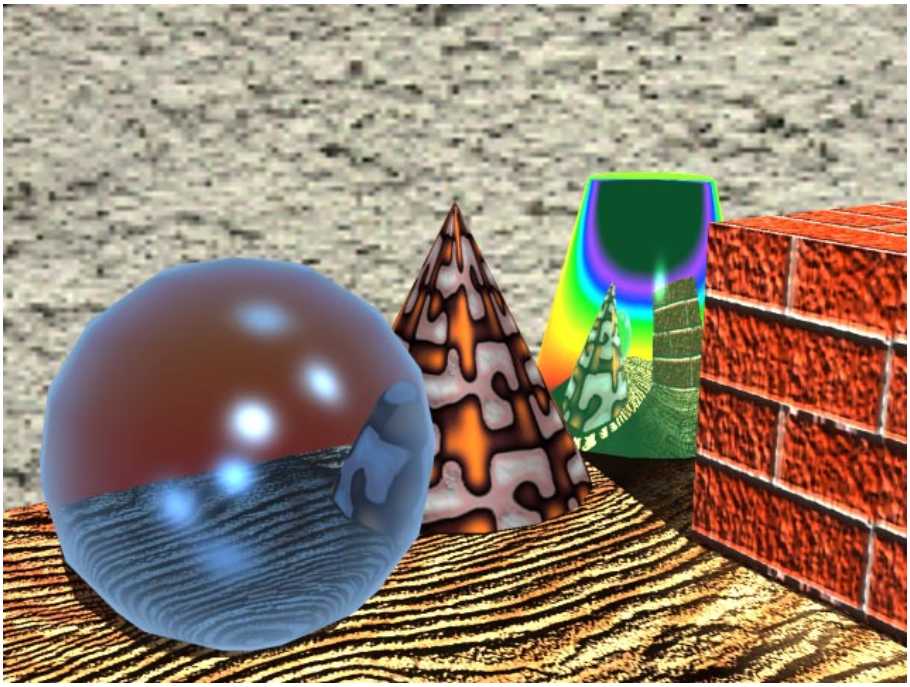


Abbildung 22.4: Szene mit 5 Objekten, gerendert im Raytracer-Modus

Kapitel 23

VRML

VRML, sprich Wörmel, ist eine für das WWW entworfene Virtual Reality Modelling Language zur Beschreibung von 3-dimensionalen Szenen mit multimedialen Komponenten und Animation. Die gerenderte Projektion der Szene kann von jedem Web-Browser betrachtet werden, der über ein passendes Plugin verfügt.

23.1 Geschichte

Bei der Nutzung von Computer-Resourcen läßt sich ein weiterer Paradigmenwechsel beobachten: Während in der EDV-Gründerzeit speziell ausgebildete Techniker am Mainframe-Computer Befehle eintippten, danach einige Jahrzehnte später Kopfarbeiter per Drag & Drop Fensterelemente manipulierten, surft inzwischen jedermann und -frau in einer weltweit vernetzten multimedialen Landschaft. Der Kontext hat sich also von institutionell über persönlich zu sozial gewandelt.

Aus diesem Zeitgeist heraus diskutierten im April 1994 auf der 1st International WWW Conference in Genf Tim Berners-Lee, einer der Väter des WWW, mit den beiden Autoren des Systems Labyrinth, Mark Pesce und Tony Parisi. Es wurde eine Mailing List aufgesetzt, die schon nach wenigen Wochen mit mehr als 1000 Teilnehmern über Syntax für Strukturen, Verhalten und Kommunikation debattierte. Bereits im Oktober 1994 wurde auf der 2nd International WWW Conference in Chicago VRML 1.0 vorgestellt, ein Entwurf, der wesentlich vom Silicon Graphics System Open Inventor inspiriert war. VRML 1.0 konnte bereits geometrische Grundkörper und Polygone in einem Koordiantensystem plazieren und ihre Farbe und Materialeigenschaften spezifizieren. Auch ließen sich durch Hyperlinks Objekte beliebig im Web referieren. Abgesehen von dieser Möglichkeit der Interaktion handelte es sich allerdings um rein statische Szenen.

Diesem Manko sollte eine schnellstens eingerichtete VAG (VRML Architecture Group) abhelfen, welche Überlegungen zur Animation und zur Integration multimedialer Komponenten wie Sound und Video koordinierte. Zur 1st International VRML Conference im Dez. 1995 war es dann soweit: Als Sieger einer Ausschreibung für VRML 97 ging Moving Worlds von Silicon Graphics nach einer On-Line-Wahl. Überarbeitete Syntax sorgte für die Beschreibung statischer Szenen mit multimedialen Bestandteilen und ein neues Event-Handling-Konzept erlaubte Animation dieser Szenen sowie Interaktion mit dem Benutzer.

23.2 Einbettung

VRML-Szenen werden beschrieben in ASCII-Dateien mit der Endung *.wrl, welche innerhalb einer HTML-Seite mit dem EMBED-Kommando referiert werden, z.B.

```
<EMBED SRC=zimmer.wrl  WIDTH=600 HEIGHT=400>
```

Ein entsprechend konfigurierter Web-Server schickt dem anfordernden Clienten als Vorspann dieser Daten den Mime-Typ VRML, worauf das zur Betrachtung installierte Plugin, z.B. Cosmo Player 2.0 von Silicon Graphics, die eingehenden Daten in eine interne Datenstruktur einliest, von wo sie zur Berechnung einer fotorealistischen Projektion verwendet werden. In welcher Weise Blickwinkel und Orientierung in der Szene modifiziert werden können, bleibt dem Plugin überlassen: Mit Mauszeiger und Keyboard Shortcuts wandert der Benutzer durch eine virtuelle Welt, verkörpert im wahrsten Sinne des Wortes durch einen Avatar, seiner geometrischen Repräsentation, beschränkt in seiner Beweglichkeit durch physikalische Restriktionen und durch eine simulierte Schwerkraft.

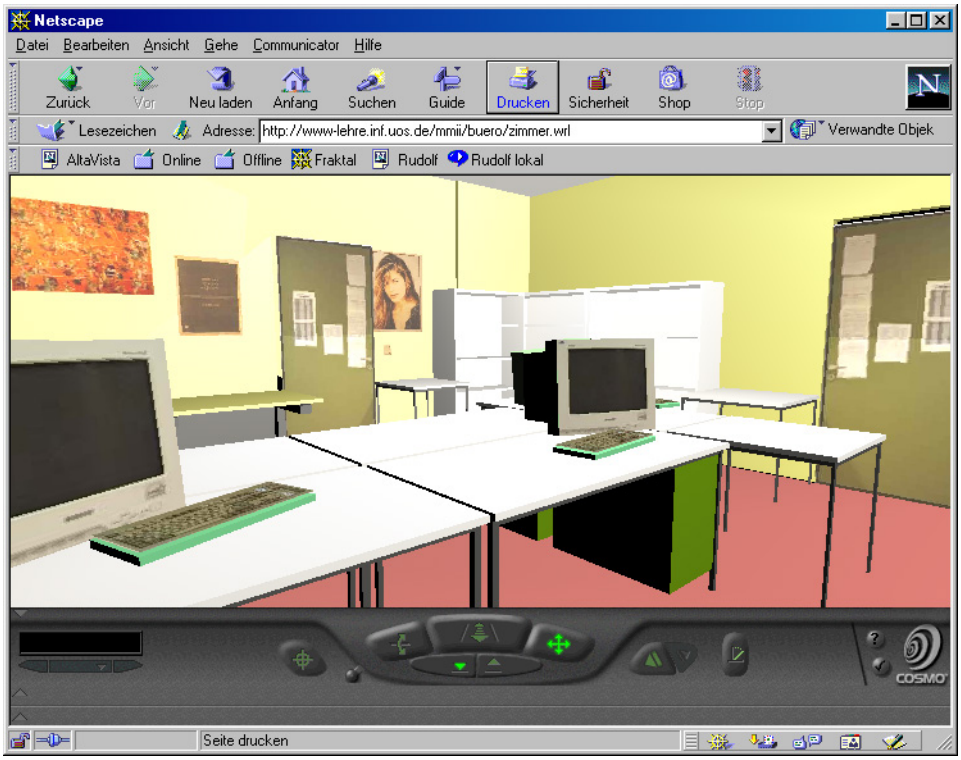


Abbildung 23.1: Screenshot vom Cosmo Player Plugin

23.3 Geometrie

Wichtigster Bestandteil von VRML-Szenen ist der sogenannte Knoten (meistens mit großem Anfangsbuchstaben geschrieben) der ähnlich eines Programmiersprachenrecords aus Feldern verschiedenen Typs besteht (meistens klein geschrieben). Diese Felder verweisen entweder auf nicht weiter strukturierte Objektknoten oder andere Gruppenknoten, die wiederum mittels ihrer Felder verzweigen können.

Beispiel 1 zeigt den Aufbau einer Szene, in der eine Kugel mit Radius 1.5 im Ursprung des Weltkoordinatensystems plaziert wird. Die x-Richtung entspricht der horizontalen Bewegung, y beschreibt die vertikale Richtung und z wächst auf den Betrachter zu. Der Sphere-Knoten hat dabei als einziges (optionales) Feld den Radius. Diese Kugel wird referiert über das geometry-Feld des Shape-Knotens, zuständig für die Gestaltung eines Objekts. Über das appearance-Feld wird die Materialbeschaffenheit in Form einer RGB-Farbe und eines Reflexions-Koeffizienten spezifiziert. Der Shape-Knoten wiederum ist als eins der Kinder im Transform-Knoten eingetragen, der über ein translation-Feld für die Verschiebung der Kugel sorgt.

```
#VRML V2.0 utf8
# kugel.wrl:
# gruene, stark reflektierende  Kugel mit Radius 1.5

Transform {                         # Plaziere
  translation 0 0 0                 # im Ursprung
  children [
    Shape {                         # eine Gestalt
      geometry Sphere {             # von der Form einer Kugel
        radius 1.5                  # mit Durchmesser 1.5
      }
      appearance Appearance {       # in der Erscheinung
        material Material {         # mit Materialbeschaffenheit
          diffuseColor 0 1 0        # gruene Farbe
          shininess 0.9             # stark reflektierend
        }
      }
    }
  ]
}
```

kugel.wrl

23.4 Polygone

Neben den Grundbausteinen Sphere (Kugel), Box (Quader), Cone (Kegel) und Cylinder (Zylinder) lassen sich eigene geometrische Gebilde konstruieren. Ausgehend von einer Liste von 3-D-Punkten im Raum werden jeweils gegen den Uhrzeigersinn alle Punkte durchlaufen, die ein Face (durch Polygon approximierte Körperfläche) aufspannen. Beispiel 2 zeigt die Definition einer 5-farbigen Pyramide mit quadratischer Grundfläche.

```
#VRML V2.0 utf8
# pyramide.wrl:
# selbstdefinierte 5-seitige Pyramide

Shape {
  geometry IndexedFaceSet {

    coord Coordinate {
      point [            # beteiligte Punkte
         0  3  0         # 0. Pyramidenpunkt (Spitze)
         0  0 -2         # 1. Pyramidenpunkt (Norden)
        -2  0  0         # 2. Pyramidenpunkt (Westen)
         0  0  2         # 3. Pyramidenpunkt (Sueden)
         2  0  0         # 4. Pyramidenpunkt (Osten )
      ]
    }

    coordIndex [         # Polygone gegen Uhrzeiger, Ende: -1
         4  3  2  1 -1   # 0. Face: Punkte 4 3 2 1 (Grundflaeche)
         0  1  2    -1   # 1. Face: Punkte 0 1 2   (Nordwesten)
         0  2  3    -1   # 2. Face: Punkte 0 2 3   (Suedwesten)
         0  3  4    -1   # 3. Face: Punkte 0 3 4   (Suedosten)
         0  4  1         # 4. Face: Punkte 0 4 1   (Nordosten)
    ]

    colorPerVertex FALSE
    color Color {
        color [          # pro Face eine Farbe benennen
          0 1 1          # 0. Face: Cyan
          1 0 0          # 1. Face: Rot
          1 1 0          # 2. Face: Gelb
          0 1 0          # 3. Face: Gruen
          0 0 1          # 4. Face: Blau
        ]
    }
  }
}
```

pyramide.wrl

23.5 Wiederverwendung

Zur Reduktion der Dateigrößen und zur besseren Lesbarkeit lassen sich einmal spezifizierte Welten wiederverwenden. Beispiel 3 zeigt die Kombination der beiden graphischen Objekte Kugel und Pyramide, wobei die Pyramide leicht nach hinten gekippt oberhalb der Kugel positioniert wird. Ferner wurde ein Hyperlink eingerichtet, der zu einer weiteren VRML-Welt führt, sobald der Mauszeiger auf der Kugel gedrückt wird.

```
#VRML V2.0 utf8
# gruppe.wrl:
# Kugel mit Hyperlink unter gekippter Pyramide

Transform{                              # Plaziere
  children[                             # die folgenden Objekte:

    Anchor {                            # Hyperlink
      url "multimedia.wrl"              # zur VRML-Datei textur.wrl
      description "Next world"          # Anzeige im Statusfenster
      children[                         # zugeordnet ist die Welt
        Inline {url "kugel.wrl"}        # aus Datei kugel.wrl
      ]
    }

    Transform {
      translation  0 1   0              # hebe 1 Einheit nach oben
      scale        1 0.5 1              # skaliere bzgl. y auf 50 %
      rotation     1 0   0 -0.523333    # kippe um 30 Grad nach hinten
      children[                         # verwende Welt
        Inline {url "pyramide.wrl"}     # aus Datei pyramide.wrl
      ]
    }

  ]
}
```

gruppe.wrl

23.6 Multimedia

Neben geometrischen Strukturen können VRML-Szenen auch multimediale Bestandteile wie Bilder, Audio und Video enthalten.

Beispiel 4 zeigt einen Würfel, auf den ein jpg-Bild als Textur aufgebracht wurde. Einem Sound-Knoten ist per URL eine Midi-Datei mit Position und Schallrichtung zugeordnet.

Sobald der Betrachter bei Annäherung an den Würfel eine gewissen Grenzwert überschritten hat, beginnt der Sound-Knoten mit dem Abspielen der Musik.

```
#VRML V2.0 utf8
# multimedia.wrl:
# Quader mit Bild-Textur und Soundquelle

Shape {                             # ein Gestaltknoten
  geometry Box {size 1 1 1}         # ein Quader der Kantenlaenge 1
  appearance Appearance {           # mit dem Aussehen
    texture ImageTexture {          # einer Bild-Textur
      url "posaune.jpg"             # aus der JPEG-Datei posaune.jpg
    }
  }
}

Sound {                             # ein Soundknoten
  source AudioClip {                # gespeist von einem Audio-Clip
    url  "ragtime.mid"              # aus der Midi-Datei ragtime.mid
    loop TRUE                       # in einer Endlosschleife
  }

  location  0 0 0                   # Schallquelle im Ursprung
  direction 0 0 1                   # dem Betrachter zugewandt
  minFront 1                        # Hoerbereichsanfang
  maxFront 8                        # Hoerbereichsende
}
```

multimedia.wrl

23.7 Interaktion

VRML97 bietet zahlreiche Möglichkeiten, mit denen einer Szene Dynamik und Interaktion verliehen werden kann. Die zentralen Bausteine für die hierzu erforderliche Ereignisbehandlung sind die EventIn- bzw. EventOut-Felder von Knoten, mit denen Meldungen empfangen und Zustandsänderungen weitergeschickt werden können. Es gibt Time,- Proximity-, Visibility-, Collision- und Touch-Sensoren, welche das Verstreichen einer Zeitspanne, das Annähern des Benutzers, die Sichtbarkeit von Objekten, das Zusammentreffen des Avatars mit einem Objekt und die Berührung mit dem Mauszeiger signalisieren. Verständlicherweise müssen Typ des verschickenden Ereignisfeldes und Typ des empfangenden Ereignisfeldes übereinstimmen.

Beispiel 5 zeigt die Kugel versehen mit einem Touch-Sensor, welcher bei Mausdruck eine Nachricht an den Soundknoten schickt, der auf diese Weise seinen Spielbeginnzeitpunkt erhält und die zugeordnete Wave-Datei startet.

```
#VRML V2.0 utf8
# interaktion.wrl:
# Kugel macht Geraeusch bei Beruehrung

Group {                                 # plaziere Gruppenknoten
  children [                            # bestehend aus
    DEF Taste TouchSensor {}            # einem Touch-Sensor
    Inline { url "kugel.wrl" }          # und einer Kugel
  ]
}

Sound {                                 # plaziere Soundknoten
  source DEF Tut AudioClip {            # gespeist von Audio-Clip
    url "tut.wav"                       # aus der Wave-Datei tut.wav
  }
  minFront  5                           # Anfang des Schallbereichs
  maxFront 50                           # Ende des Schallbereichs
}

ROUTE Taste.touchTime                   # bei Beruehrung der Kugel
   TO Tut.set_startTime                 # schicke Systemzeit an den Knoten Tut
```

interaktion.wrl

23.8 Animation

Die benutzergesteuerte oder automatische Bewegung von Objekten und Szenenteilen wird wiederum mit Hilfe der Ereignisbehandlung organisiert. Im Beispiel 6 wird die Ziehbewegung des gedrückten Mauszeigers zur Manipulation der lokalen Translation eines Objekts verwendet und das regelmäßige Verstreichen eines Zeitintervalls löst eine Nachricht an denselben geometrischen Knoten aus, der hierdurch seinen aktuellen Rotationsvektor erhält. Eine solche Konstruktion verlangt einen Orientation-Interpolator, welcher beliebige Bruchzahlen zwischen 0 und 1 auf die zugeordneten Werte seines Schlüsselintervalls abbildet, hier bestehend aus allen Drehwinkeln zwischen 0 und 3.14 (=180 Grad beschrieben in Bogenmaß), bezogen auf die y-Achse.

```
#VRML V2.0 utf8
# animation.wrl:
# selbstaendig sich drehende und interaktiv verschiebbare Pyramide

DEF Schieber PlaneSensor {}            # Sensor zum Melden einer Mausbewegung

DEF Timer TimeSensor {                 # Sensor zum Melden eines Zeitintervalls
  cycleInterval 5                      # Dauer 5 Sekunden
  loop TRUE                            # Endlosschleife
}

DEF Rotierer OrientationInterpolator{  # Interpolator fuer Rotation
  key      [0 , 1]                     # bilde Schluessel 0 und 1 ab auf
  keyValue [ 0  1  0  0                #   0 Grad Drehung bzgl. y
             0  1  0  3.14]            # 180 Grad Drehung bzgl. y
}

DEF Pyramide Transform {               # plaziere Objekt mit Namen Pyramide
      children [                       # bestehend aus
        Inline {url "pyramide.wrl"}    # VRML-Datei pyramide.wrl
      ]
}

ROUTE Timer.fraction_changed           # falls Zeitintervall sich aendert
   TO Rotierer.set_fraction            # schicke Bruchteil an Rotierer

ROUTE Rotierer.value_changed           # falls Drehung sich aendert
   TO Pyramide.set_rotation            # schicke Drehwert an Pyramide

ROUTE Schieber.translation_changed     # falls gedrueckter Mauszeiger bewegt wird
   TO Pyramide.set_translation         # schicke Translationswert an Pyramide
```

animation.wrl

23.9 Scripts

Manchmal reichen die in VRML angebotenen Funktionen wie Sensoren und Interpolatoren nicht aus, um ein spezielles situationsbedingtes Interaktionsverhalten zu erzeugen. Abhilfe schafft hier der sogenannte Script-Knoten, welcher Input empfangen, Daten verarbeiten und Output verschicken kann. Z.B. kann eine vom Touch-Sensor geschickte Nachricht eine Berechnung anstoßen, deren Ergebnis in Form einer Translations-Nachricht an ein bestimmtes Objekt geschickt und dort zur Neupositionierung genutzt wird.

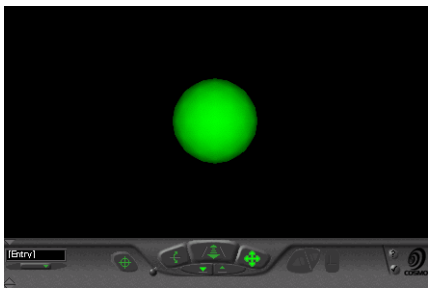
```
#VRML V2.0 utf8
# javascript.wrl:
# Rotation eines Objekts wird ueber Javascript manipuliert

Group {                                 # gruppiere
  children [                            # folgende Objekte
    DEF Taste TouchSensor{}             # Beruehrungssensor Taste
    DEF Pyramide Transform {            # Objekt Pyramide
      children[                         # bestehend aus
        Inline {url "pyramide.wrl"}     # VRML-Welt pyramide.wrl
      ]
    }
  ]
}

DEF Aktion Script {                     # Script mit Namen Aktion
  eventIn  SFBool     isActive          # Input-Parameter
  eventOut SFRotation drehung           # Output-Parameter
  url [                                 # gespeist von
     "javascript:                      // inline-Javascript
     function isActive(eventValue) {   // fuer eventIn zustaendig
       if (eventValue == true) {       // falls eventValue den Wert wahr hat
         drehung[0] = 0.0;             // drehe
         drehung[1] = 1.0;             // bzgl.
         drehung[2] = 0.0;             // der y-Achse
         drehung[3] += 0.174444;       // um weitere 10 Grad
       }
     }"
  ]
}

ROUTE Taste.isActive                    # bei Beruehren der Pyramide
   TO Aktion.isActive                   # sende Nachricht an das Script Aktion

ROUTE Aktion.drehung                    # vom Script Aktion erzeugter Dreh-Vektor
   TO Pyramide.set_rotation             # wird an die Pyramide geschickt
```

javascript.wrl

Die Formulierung des Berechnungsalgorithmus geschieht entweder durch ein Javascript-Programm,

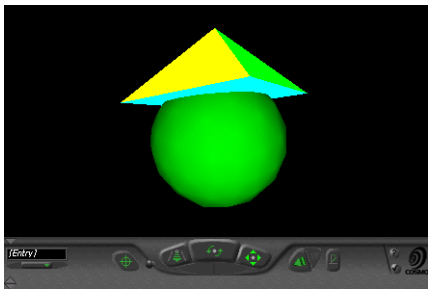
inline gelistet im Script-Knoten, oder durch eine assoziierte Java-Klasse, welche in übersetzter Form mit Dateiendung *.class lokal oder im Netz liegt. Zum Übersetzen der Java-Quelle ist das EAI (External Authoring Interface) erforderlich, welches in Form einiger Packages aus dem Verzeichnis importiert wird, in welches sie das VRML-Plugin bei der Installation deponiert hatte.

Beispiel 7 zeigt die Pyramide zusammen mit einem Java-Script, welches bei jedem Aufruf den Drehwinkel bzgl. der y-Achse um weitere 10 Grad erhöht.



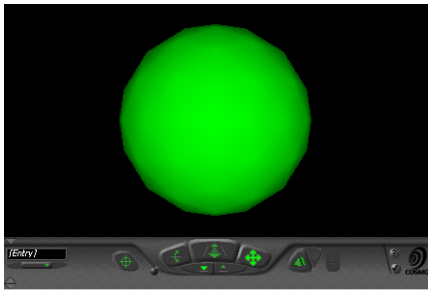
kugel.wrl

pyramide.wrl



gruppe.wrl

multimedia.wrl



interaktion.wrl

animation.wrl

Abbildung 23.2: Screenshots der Beispiele 1 - 6

23.10 Multiuser

Eines der ursprünglichen Entwicklungsziele von VRML bleibt auch bei VRML 97 offen: es gibt noch keinen Standard für Multiuser-Welten. Hiermit sind Szenen gemeint, in denen mehrere Benutzer gleichzeitig herumwandern und interagieren können. Da jedes ausgelöste Ereignis von allen Beteiligten wahrgenommen werden soll, muß ein zentraler Server den jeweiligen Zustand der Welt verwalten und den anfragenden Klienten fortwährend Updates schicken. In diesem Zusammenhang erhält der Avatar eine aufgewertete Rolle: Zusätzlich zu seiner geometrischen Räumlichkeit, die schon zur Kollisionsdetektion in einer Single-User-Szenerie benötigt wurde, muß nun auch sein visuelles Äußeres spezifiziert werden, sicherlich ein weiterer wichtiger Schritt zur Verschmelzung eines real existierenden Benutzers mit der von ihm gespielten Rolle.

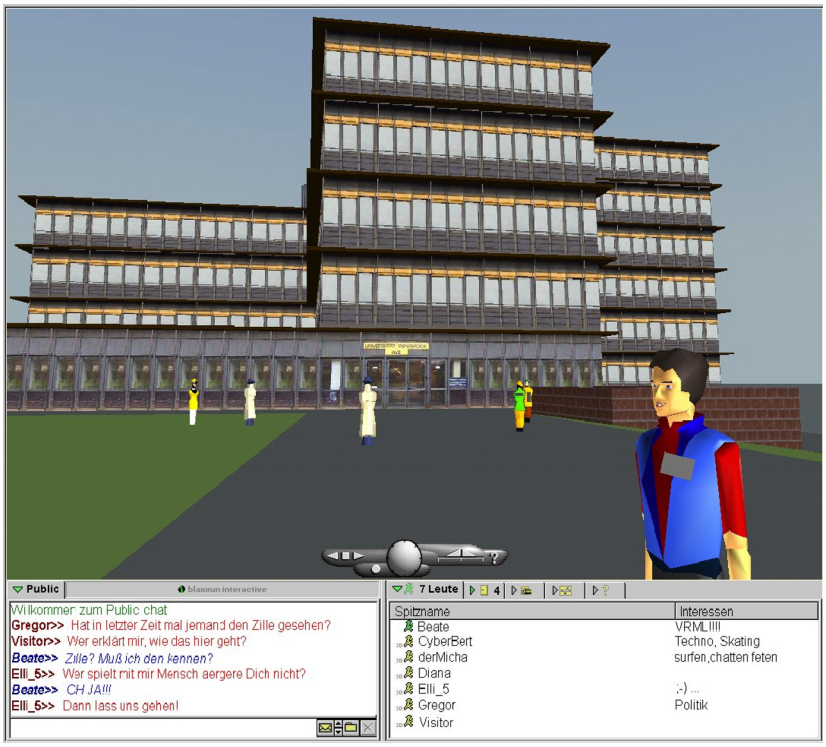


Abbildung 23.3: Screenshot vom Blaxxun Contact Plugin

Unter der Einstiegsseite `http://www-lehre.inf.uos.de/avz` kann das *Virtuelle AVZ* der Universität Osnabrück betreten werden, welches mit Hilfe eines Blaxxun-Community-Servers als Multiuser-Welt realisiert wurde.

Kapitel 24

OpenGL

24.1 Grundlagen

OpenGL bietet eine Schnittstelle zwischen dem Anwenderprogramm und der Grafikhardware eines Computers zum Modellieren und Projizieren von dreidimensionalen Objekten.

Diese Schnittstelle enthält

- 200 Befehle in der *OpenGL Library* (beginnen mit `gl`) zum Verwalten von elementaren geometrischen Primiviten wie Punkte, Linien, Polygone, Bezierkurven und ihrer Attribute wie Farben und Normalenvektoren.
- 50 Befehle in der *OpenGL Utility Library* (beginnen mit `glu`) zum Verwalten von *NURBS* (non uniform rational b-splines) und *quadrics* (Körper mit durch quadratische Gleichungen beschreibbaren Oberflächen, z.B. Kugel, Zylinder) sowie zum vereinfachten Manipulieren von Projektionsmatrizen.

OpenGl enthält keine Routinen zur Benutzerein- und -ausgabe und zur Kommunikation mit dem (plattformspezifischen) Fenstersystem. Hierfür sind zuständig

- 30 Befehle im *OpenGL Utility Toolkit* (beginnen mit `glut`) zur Anbindung der von OpenGl gerenderten Ausgabe an das jeweilige Fenstersystem und zum Verwalten von höheren geometrischen Objekten wie Kugel, Kegel, Torus und Teapot.

Bei manchen OpenGl-Implementationen (z.B. X Window System) kann die Berechnung der Grafik und ihre Ausgabe auf verschiedenen Rechnern stattfinden. Der *Klient* ruft hierbei ein OpenGl-Kommando auf, mit Hilfe eines *Protokolls* wird es übertragen und der *Server* setzt es in ein Bild um.

Um die Kommunikationsbandbreite zu minimieren, arbeitet OpenGl als Zustandsmaschine. Dies bedeutet, daß einmal gesetzte Zustände (z.B. die Vordergrundfarbe) bis zu ihrem Widerruf beibehalten werden. Auch können komplexe geometrische Figuren unter einem Namen in einer *Display-Liste* eingetragen werden, so daß sie im Falle einer Wiederverwendung nicht erneut übertragen werden müssen sondern unter Angabe ihres Namens ausgewertet werden können.

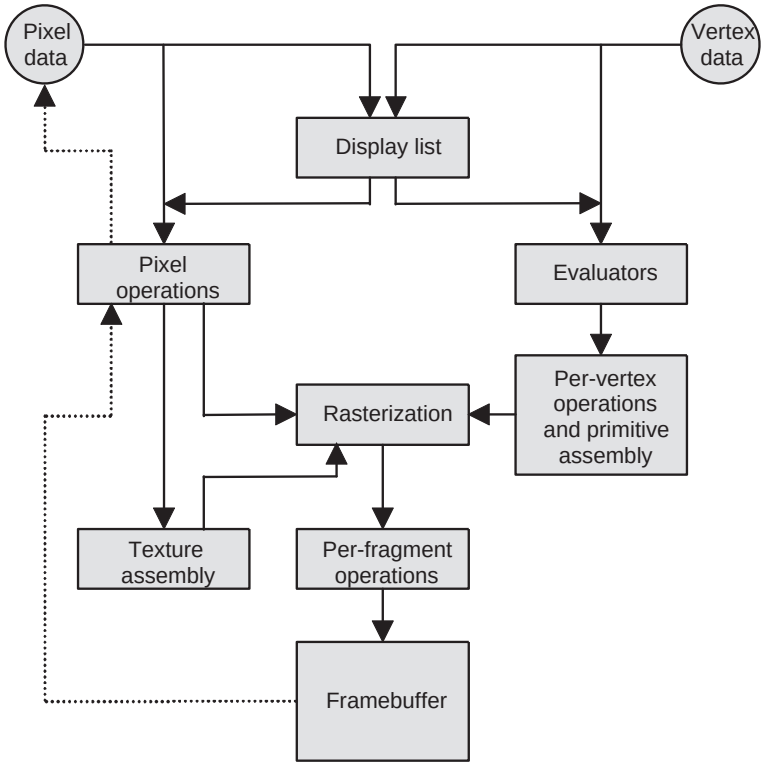


Abbildung 24.1: OpenGl Rendering Pipeline

Die Bestandteile der OpenGl Rendering Pipeline lauten:

- *Vertex data* (Knoten) oder *Pixel data* (Pixel) sind definierende Bestandteile eines zu berechnenden Bildes.
- *display lists* referenzieren Vertex und Pixel data.
- *Pixel operations* regeln das Pre- und Postprocessing von Pixeldaten.
- *evalutators* approximieren die durch Kontrollpunkte definierten parametrischen Kurven und Oberflächen zu Polygonzügen und Flächen samt Normalenvektoren.
- *Per Vertex operations* transformieren 3D-Welt-Koordinaten nach Vorgabe der syntetischen Kamera.
- *Primitive Assembly* erledigt clipping und backface removal.
- *Texture Assembly* bildet benutzerdefinierte Texturen auf Objekte ab.
- *Rasterization* überführt geometrische Daten und Pixelmengen zu *fragments*, welche die für eine Bildschirmkoordinate relevanten Informationen aufsammeln.
- *per fragment operations* bilden den Inhalt eines Fragments unter Berücksichtigung des Tiefenpuffers und unter Anwendung von Nebel- und Alpha-Masken auf eine Bildschirmkoordinate im *Frame buffer* ab.

24.2 Syntax

Durch Aufruf von Prozeduren manipuliert der Benutzer seine Daten. OpenGl verwendet dabei acht Datentypen. Der in Spalte 1 von Tabelle 24.1 genannte Suffix kündigt als letzter Buchstabe eines Kommandos den erwarteten Datentyp an.

Suffix	Datentyp	C-Korrespondenz	OpenGl Name
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int	GLint
f	32-bit floating point	float	GLfloat, GLclampf
d	64-bit floating point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int	GLuint, GLenum, GLbitfield

Tabelle 24.1: OpenGL Datentypen

Da OpenGl-Kommandos keine Überladung kennen, müssen für alle sinnvollen Parameterdatentypen unterschiedliche Prozedurnamen vorhanden sein.

Zum Beispiel erzeugt jeder der beiden Befehle

```
glVertex2i(5,3);
glVertex2f(5.0, 3.0);
```

einen zweidimensionalen Punkt mit x-Koordinate 5 und y-Koordinate 3.

Manche OpenGl-Kommandos enden mit dem Buchstaben **v** und signalisieren dadurch, daß sie einen Vektor als Übergabeparameter erwarten. Zum Beipiel kann das Setzen eines RGB-Farbwerts durch Angabe von drei Zahlen oder durch Angabe eines drei-elementigen Vektors geschehen:

```
glColor3f(1.0, 0.8, 0.8);
GLfloat farbvektor[] = {1.0, 0.8, 0.8};
glColor3fv(farbvektor);
```

Elementare geometrischen Figuren (wie z.B. Linien, Dreiecke, Vierecke, Polygone, etc.) werden mittels `glVertex*` durch eine Folge von Punkten $p_0, p_1, p_2, ..., p_{n-1}$ definiert, welche durch zwei Kommandos geklammert wird:

```
glBegin(MODUS);
...
glEnd();
```

Hierbei ist `MODUS` eine der folgenden vordefinierten OpenGl-Konstanten:

`GL_POINTS`	Punkte $p_0, p_1, p_2, ...$
`GL_LINES`	Gradenstücke $(p_0, p_1), (p_2, p_3), ...$
`GL_LINE_STRIP`	Linienzug $(p_0, p_1, p_2, p_3, ..., p_{n-1})$
`GL_LINE_LOOP`	geschlossener Linienzug $(p_0, p_1, p_2, p_3, ..., p_{n-1}, p_0)$
`GL_TRIANGLES`	Dreiecke $(p_0, p_1, p_2), (p_3, p_4, p_5), (p_6, p_7, p_8), ...$
`GL_TRIANGLE_STRIP`	Streifen von Dreiecken $(p_0, p_1, p_2), (p_2, p_1, p3), (p_2, p_3, p_4), ...$
`GL_TRIANGLE_FAN`	Fächer von Dreiecken $(p_0, p_1, p_2), (p_0, p_2, p_3), (p_0, p_3, p_4), ...$
`GL_QUADS`	Vierecke $(p_0, p_1, p_2, p_3), (p_4, p_5, p_6, p_7), (p_8, p_9, p_10, p_11), ...$
`GL_QUAD_STRIP`	Streifen von Vierecken $(p_0, p_1, p_3, p_2), (p_2, p_3, p_5, p_4), (p_4, p_5, p_7, p_6), ...$
`GL_POLYGON`	konvexes (!) Polygon $(p_0, p_1, p_2, ..., p_0)$

Es existieren Stacks von 4×4 Matrizen zur Transformation des Modells und zur Transformation der Projektion.

Die Auswahl geschieht durch `glMatrixMode()`. Durch die Anweisung

```
glMatrixMode(GL_PROJECTION);
```

bzw. durch die Anweisung

```
glMatrixMode(GL_MODEL_VIEW);
```

beziehen sich bis auf weiteres alle Matrix-Operationen auf den Projektions-Stack bzw. auf den Modell-Stack.

Zum Sichern und späteren Wiederherstellen der aktuellen Transformationsmatrix wird diese durch den Befehl `glPushMatrix()` kopiert und oben auf den Stack gelegt. Nun kann sie beliebig manipuliert werden. Ihr Originalzustand kann dann später durch `glPopMatrix()` wiederhergestellt werden.

Multiplikation der obersten Stackmatrix mit einer beliebigen Matrix m geschieht durch Definition der Matrix durch Aufruf der Multiplikation:

```
GLfloat m[][] = {{2.0,0.0,0.0,0.0},
                 {0.0,4.0,0.0,0.0},
                 {0.0,0.0,3.0,0.0},
                 {0.0,0.0,0.0,2.0}};
glMultMatrix(m);
```

24.3 Programmbeispiele

Auf den folgenden Seiten werden einige durch deutsche Kommentare erweiterte OpenGl-Beispiele vorgestellt, die alle dem bei *Addison-Wesley* erschienenen Buch *OpenGL Programming Guide* entnommen wurden. Das Copyright für diese Programme liegt bei der Firma *Silicon Graphics*. Aus Platzgründen wurde auf eine explizite Wiederholung der Copyright-Klausel im jeweiligen Quelltext verzichtet. Eine komplette Sammlung aller Beispiele findet sich unter `http://trant.sgi.com/opengl/examples/redbook/redbook.html` .

hello.c zeigt in Orthogonalprojektion ein rotes Rechteck auf blauem Hintergrund. Es wird zunächst um eine Einheit in x-Richtung verschoben und danach um 45 Grad gedreht.

wire-cube.c zeigt in perspektivischer Projektion einen roten Drahtgitterwürfel vor grünem Hintergrund. Der Würfel ist in y-Richtung um den Faktor 1.5 skaliert und um 30 Grad bzgl. der y-Achse gedreht.

wire-prop-cube.c zeigt in perspektivischer Projektion einen weißen Drahtgittelwürfel vor blauem Hintergrund. Bei Verändern des Ausgabefensters durch den Benutzer wird die Funktion `reshape` aufgerufen, die eine Neuberechnung des Viewports durchführt.

click.c zeigt die Interaktion durch die Maus für den Farbwechsel eines Dreiecks. Die globale Variablen g wird durch den linken Maus-Button auf 1, durch den rechten Mausbutton auf 0 gesetzt. In der Funktion *display* wird dadurch entweder ein gelb oder ein rot erzeugt.

key.c zeigt die Interaktion durch die Tastatur für den Farbwechsel eines Dreiecks. Die globalen Variablen *r,g* und *b* werden durch Betätigen der gleichnamigen Tasten entsprechend gesetzt. In der Funktion *display* wird dadurch das RGB-Tripel auf *rot, grün* oder *blau* gesetzt.

planet.c zeigt die Interaktion durch die Tastatur für das Fortschalten einer Rotationsbewegung. Durch Drücken der *d*-Taste wird die Rotation der Erde um sich selbst um 10 Grad angestoßen, durch Drücken der *y*-Taste wird die Rotation der Erde um die Sonne um 5 Grad angestoßen.

smooth.c zeigt ein Dreieck mit Farbverlauf. Erreicht wird dies im Smooth-Shading-Modus durch Interpolation von RGB-Werten, welche an den Ecken des Dreiecks vorgegeben sind.

list.c zeigt eine Anwendung von Display-Listen. Unter dem Namen *listName* wird ein rotes, gefülltes Dreieck zusammen mit einer Translation hinterlegt. Ein wiederholter Aufruf dieses Namens erzeugt eine Sequenz von nebeneinander platzierten Dreiecken.

bezier-curve.c zeigt eine durch vier zwei-dimensionale Kontrollpunkte definierte Bezier-Kurve. Zur besseren Sichtbarkeit werden die Kontrollpunkte mit 5-facher Stärke platziert. Die Kurve entsteht durch Auswertung eines Evaluators längs eines von 0 nach 1 in 30 Schritten wandernden Parameters.

bezier-surface.c zeigt ein durch 16 drei-dimensionale Kontrollpunkte definiertes Bezier-Gitter. Das Gitter entsteht durch Auswertung eines Evaluators längs eines von 0 nach 1 in 8 Schritten und von 0 nach 1 in 30 Schritten wandernden Parameters.

teapot.c zeigt den klassischen Utah-Teapot in photo-realistischer Projektion. Berücksichtigt werden Reflektionskoeffizienten für diffuses und spekulares Licht, erzeugt von einer Lichtquelle.

teapot-rotate.c zeigt den klassischen Utah-Teapot in Bewegung. Erreicht wird dies durch die Funktion *spinDisplay*, welche die Rotation um die y-Achse in 5-Grad-Schritten fortschreibt. *spinDisplay* wird durch die Registrierung über *glutIdleFunc* immer dann aufgerufen, wenn keine anderen Events zur Abarbeitung anstehen.

```
#include <GL/glut.h>                          /* Header fuer OpenGl utility toolkit  */
#include <stdlib.h>                           /* Header fuer C-Library               */

void init(void) {                             /* Initialisierung                     */
  glClearColor(0.0, 0.0, 1.0, 0.0);           /* setze Blau als Hintergrundfarbe     */
  glMatrixMode(GL_PROJECTION);                /* betrifft Projektionsmatrix          */
  glLoadIdentity();                           /* beginne mit Einheitsmatrix          */
  glOrtho(0.0, 3.0, 0.0, 2.0, -1.0, 1.0);     /* Orthogonalprojektions-Clip-Ebenen   */
}                                             /* left, right, bottom, top, near, far */

void display(void){                           /* Prozedur zum Zeichnen               */
  glClear(GL_COLOR_BUFFER_BIT);               /* alle Pixel zurecksetzen             */
  glMatrixMode(GL_MODELVIEW);                 /* betrifft Modelview-Matrix           */
  glLoadIdentity();                           /* beginne mit Einheitsmatrix          */
  glRotatef(45,0.0,0.0,1.0);                  /* drehe 45 Grad bzgl. der z-Achse     */
  glTranslatef(1.0, 0.0, 0.0);                /* verschiebe eine Einheit nach rechts */
  glColor3f(1.0, 0.0, 0.0);                   /* Farbe rot                           */
  glBegin(GL_POLYGON);                        /* Beginn eines Polygons               */
    glVertex3f(0.25, 0.25, 0.0);              /* links unten                         */
    glVertex3f(0.75, 0.25, 0.0);              /* rechts unten                        */
    glVertex3f(0.75, 0.75, 0.0);              /* rechts oben                         */
    glVertex3f(0.25, 0.75, 0.0);              /* links oben                          */
  glEnd();                                    /* Ende des Polygons                   */
  glFlush();                                  /* direkt ausgeben                     */
}

int main (int argc, char ** argv) {           /* Hauptprogramm                       */
  glutInit(&argc, argv);                      /* initialisiere GLUT                  */
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); /* single buffer, true color           */
  glutInitWindowSize(750, 500);               /* initiale Fenstergroesse             */
  glutInitWindowPosition(100,100);            /* initiale Fensterposition            */
  glutCreateWindow("hello");                  /* Fenster mit Aufschrift              */
  init();                                     /* rufe init auf                       */
  glutDisplayFunc(display);                   /* registriere display                 */
  glutMainLoop();                             /* beginne Event-Schleife              */
  return 0;                                   /* ISO C verlangt Rueckgabe            */
}
```

hello.c: Rotes Quadrat, gedreht und verschoben, auf blauem Hintergrund

```
#include <GL/glut.h>                          /* Header fuer OpenGl utility toolkit */
#include <stdlib.h>                           /* Header fuer C-Library              */

void init(void) {                             /* Initialisierung                    */
  glClearColor(0.0, 0.0, 1.0, 0.0);           /* setze Blau als Hintergrundfarbe    */
  glMatrixMode(GL_PROJECTION);                /* ab jetzt: Projektionsmatrix        */
  glLoadIdentity();                           /* lade Einheitsmatrix                */
  glFrustum(-1.0,1.0,-1.0,1.0,1.5,20.0);      /* left,right,bottom,top,near,far     */
}

void display(void){                           /* Prozedur zum Zeichnen              */
  glClear(GL_COLOR_BUFFER_BIT);               /* alle Pixel zurecksetzen            */
  glMatrixMode(GL_MODELVIEW);                 /* ab jetzt:  Modellierungsmatrix     */
  glColor3f(1.0, 1.0, 1.0);                   /* Farbe rot                          */
  glLoadIdentity();                           /* Einheitsmatrix                     */
  gluLookAt(0.0, 0.0, 5.0,                    /* Kamera-Standpunkt                  */
            0.0, 0.0, 0.0,                    /* Kamera-Fokussiertpunkt             */
            0.0, 1.0, 0.0);                   /* Up-Vektor                          */
  glScalef(1.0, 1.5, 1.0);                    /* Skalierung in 3 Dimensionen        */
  glRotatef(30.0, 0.0, 1.0, 0.0);             /* rotiere 30 Grad um y-Achse         */
  glutWireCube(2.0);                          /* Drahtgitterwuerfel                 */
  glFlush();                                  /* direkt ausgeben                    */
}

int main (int argc, char ** argv) {           /* Hauptprogramm                      */
  glutInit(&argc, argv);                      /* initialisiere GLUT                 */
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);  /* single buffer, true color          */
  glutInitWindowSize(750, 500);               /* initiale Fenstergroesse            */
  glutInitWindowPosition(100,100);            /* initiale Fensterposition           */
  glutCreateWindow("wire-cube");              /* Fenster mit Aufschrift             */
  init();                                     /* rufe init auf                      */
  glutDisplayFunc(display);                   /* registriere display                */
  glutMainLoop();                             /* beginne Event-Schleife             */
  return 0;                                   /* ISO C verlangt Rueckgabe           */
}
```

wire-cube.c: Perspektivische Projektion eines Drahtgitterquaders, ohne Reshape

```
#include <GL/glut.h>                          /* Header fuer OpenGl utility toolkit */
#include <stdlib.h>                           /* Header fuer C-Library              */

void init(void) {                             /* Initialisierung                    */
  glClearColor(0.0, 0.0, 1.0, 0.0);           /* setze Blau als Hintergrundfarbe    */
  glMatrixMode(GL_PROJECTION);                /* ab jetzt: Projektionsmatrix        */
  glLoadIdentity();                           /* lade Einheitsmatrix                */
  glFrustum(-1.0,1.0,-1.0,1.0,1.5,20.0);      /* left,right,bottom,top,near,far     */
}

void display(void){                           /* Prozedur zum Zeichnen              */
  glClear(GL_COLOR_BUFFER_BIT);               /* alle Pixel zurecksetzen            */
  glMatrixMode(GL_MODELVIEW);                 /* ab jetzt:  Modellierungsmatrix     */
  glColor3f(1.0, 0.0, 0.0);                   /* Farbe rot                          */
  glLoadIdentity();                           /* Einheitsmatrix                     */
  gluLookAt(0.0, 0.0, 5.0,                    /* Kamera-Standpunkt                  */
            0.0, 0.0, 0.0,                    /* Kamera-Fokussierpunkt              */
            0.0, 1.0, 0.0);                   /* Up-Vektor                          */
  glScalef(1.0, 1.5, 1.0);                    /* Skalierung in 3 Dimensionen        */
  glRotatef(30.0, 0.0, 1.0, 0.0);             /* rotiere 30 Grad um y-Achse         */
  glutWireCube(2.0);                          /* Drahtgitterwuerfel                 */
  glFlush();                                  /* direkt ausgeben                    */
}

void reshape(int w, int h) {
  GLfloat p = (GLfloat) w / (GLfloat) h;      /* Proportionalitaetsfaktor           */
  glViewport(0, 0, (GLsizei)w, (GLsizei)h);   /* setze Viewport                     */
  glMatrixMode(GL_PROJECTION);                /* betrifft Projektionsmatrix         */
  glLoadIdentity();                           /* lade Einheitsmatrix                */
  if (p > 1.0)                                /* falls breiter als hoch             */
       glFrustum(  -p,  p,-1.0,1.0,1.5,20.0); /* left,right,bottom,top,near,far     */
  else glFrustum(-1.0,1.0,-1/p,1/p,1.5,20.0); /* left,right,bottom,top,near,far     */
}


int main (int argc, char ** argv) {           /* Hauptprogramm                      */
  glutInit(&argc, argv);                      /* initialisiere GLUT                 */
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);  /* single buffer, true color          */
  glutInitWindowSize(750, 500);               /* initiale Fenstergroesse            */
  glutInitWindowPosition(100,100);            /* initiale Fensterposition           */
  glutCreateWindow("wire-cube-prop");         /* Fenster mit Aufschrift             */
  init();                                     /* rufe init auf                      */
  glutDisplayFunc(display);                   /* registriere display                */
  glutReshapeFunc(reshape);                   /* registriere reshape                */
  glutMainLoop();                             /* beginne Event-Schleife             */
  return 0;                                   /* ISO C verlangt Rueckgabe           */
}
```

wire-prop-cube.c: Perspektivische Projektion eines Drahtgittelquaders, mit Reshape

```
#include <GL/glut.h>                          /* Header fuer OpenGl utility toolkit  */
#include <stdlib.h>                           /* Header fuer C-Library               */

GLfloat g=0.0;                                /* globale Variable                   */

void init(void) {                             /* Initialisierung                     */
  glClearColor(0.5, 0.5, 0.5, 0.0);           /* setze Grau als Hintergrundfarbe     */
  glMatrixMode(GL_PROJECTION);                /* betrifft Projektionsmatrix          */
  glLoadIdentity();                           /* beginne mit Einheitsmatrix          */
  glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);     /* Orthogonalprojektions-Clip-Ebenen   */
}                                             /* left, right, bottom, top, near, far */

void display(void){                           /* Prozedur zum Zeichnen               */
  glClear(GL_COLOR_BUFFER_BIT);               /* alle Pixel zurecksetzen             */
  glMatrixMode(GL_MODELVIEW);                 /* betrifft Modelview-Matrix           */
  glLoadIdentity();                           /* beginne mit Einheitsmatrix          */
  glColor3f(1.0,g,0.0);                       /* Farbe gemaess RGB-Tripel            */
  glBegin(GL_TRIANGLES);                      /* Beginn eines Dreiecks               */
    glVertex3f(0.25, 0.25, 0.0);              /* links unten                         */
    glVertex3f(0.75, 0.25, 0.0);              /* rechts unten                        */
    glVertex3f(0.25, 0.75, 0.0);              /* links oben                          */
  glEnd();                                    /* Ende des Dreiecks                   */
  glFlush();                                  /* direkt ausgeben                     */
}

void mouse(int button,int state,int x,int y){/* Maus-Klick bei Koordinate x,y       */
  switch(button) {                            /* analysiere den Mausevent            */
    case GLUT_LEFT_BUTTON:                    /* falls linke Taste gedrueckt         */
      if (state==GLUT_DOWN) g=1.0;            /* setzte Gruenanteil auf 1            */
      break;
    case GLUT_RIGHT_BUTTON:                   /* falls rechte Taste gedrueckt        */
      if (state==GLUT_DOWN) g=0.0;            /* setzte Gruenanteil auf 0            */
      break;
  } glutPostRedisplay();                      /* durchlaufe display erneut           */
}

int main (int argc, char ** argv) {           /* Hauptprogramm                       */
  glutInit(&argc, argv);                      /* initialisiere GLUT                  */
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); /* single buffer, true color           */
  glutInitWindowSize(350, 350);               /* initiale Fenstergroesse             */
  glutInitWindowPosition(100,100);            /* initiale Fensterposition            */
  glutCreateWindow("click");                  /* Fenster mit Aufschrift              */
  init();                                     /* rufe init auf                       */
  glutDisplayFunc(display);                   /* registriere display                 */
  glutMouseFunc(mouse);                       /* registriere mouse                   */
  glutMainLoop();                             /* beginne Event-Schleife              */
  return 0;                                   /* ISO C verlangt Rueckgabe            */
}
```

click.c: Interaktion durch Maus zur Farbwahl

```
#include <GL/glut.h>                          /* Header fuer OpenGl utility toolkit  */
#include <stdlib.h>                           /* Header fuer C-Library               */

GLfloat r=1.0, g=0.0, b=0.0;                  /* globales RGB-Tripel                 */

void init(void) {                             /* Initialisierung                     */
  glClearColor(0.5, 0.5, 0.5, 0.0);           /* setze Grau als Hintergrundfarbe     */
  glMatrixMode(GL_PROJECTION);                /* betrifft Projektionsmatrix          */
  glLoadIdentity();                           /* beginne mit Einheitsmatrix          */
  glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);     /* Orthogonalprojektions-Clip-Ebenen   */
}                                             /* left, right, bottom, top, near, far */

void display(void){                           /* Prozedur zum Zeichnen               */
  glClear(GL_COLOR_BUFFER_BIT);               /* alle Pixel zurecksetzen             */
  glMatrixMode(GL_MODELVIEW);                 /* betrifft Modelview-Matrix           */
  glLoadIdentity();                           /* beginne mit Einheitsmatrix          */
  glColor3f(r,g,b);                           /* Farbe gemaess RGB-Tripel            */
  glBegin(GL_TRIANGLES);                      /* Beginn eines Dreiecks               */
    glVertex3f(0.25, 0.25, 0.0);              /* links unten                         */
    glVertex3f(0.75, 0.25, 0.0);              /* rechts unten                        */
    glVertex3f(0.25, 0.75, 0.0);              /* links oben                          */
  glEnd();                                    /* Ende des Dreiecks                   */
  glFlush();                                  /* direkt ausgeben                     */
}

void key(unsigned char key, int x, int y){    /* Bei Tastendruck                     */
  switch(key) {                               /* analysiere den Tastendruck          */
    case 'r': r=1.0; g=0.0; b=0.0; break;     /* falls 'r': setze Tripel auf Rot     */
    case 'g': r=0.0; g=1.0; b=0.0; break;     /* falls 'g': setze Tripel auf Gruen   */
    case 'b': r=0.0; g=0.0; b=1.0; break;     /* falls 'b': setze Tripel auf Blau    */
  }
  glutPostRedisplay();
}

int main (int argc, char ** argv) {           /* Hauptprogramm                       */
  glutInit(&argc, argv);                      /* initialisiere GLUT                  */
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); /* single buffer, true color           */
  glutInitWindowSize(350, 350);               /* initiale Fenstergroesse             */
  glutInitWindowPosition(100,100);            /* initiale Fensterposition            */
  glutCreateWindow("key");                    /* Fenster mit Aufschrift              */
  init();                                     /* rufe init auf                       */
  glutDisplayFunc(display);                   /* registriere display                 */
  glutKeyboardFunc(key);                      /* registriere key                     */
  glutMainLoop();                             /* beginne Event-Schleife              */
  return 0;                                   /* ISO C verlangt Rueckgabe            */
}
```

key.c: Interaktion durch Tastatur zur Farbwahl

```
#include <GL/glut.h>                                  /* OpenGl Utility Toolkit      */
#include <stdlib.h>                                   /* C-library                   */

int year = 0, day = 0;                                /* Variablen fuer Drehungen    */

void init(void) {
   glClearColor (0.0, 0.0, 1.0, 0.0);                 /* blauer Hintergrund          */
   glShadeModel (GL_FLAT);                            /* Flatshading                 */
   glMatrixMode (GL_PROJECTION); glLoadIdentity ();   /* ab jetzt Projektion         */
   gluPerspective(60.0, 1.5, 1.0, 20.0);              /* Blickwinkel, w/h, near, far */
   glMatrixMode(GL_MODELVIEW); glLoadIdentity();      /* ab jetzt Modelview          */
   gluLookAt (0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);  /* Kamera, Fokussier, Up-Vektor*/
}

void display(void){
   glClear (GL_COLOR_BUFFER_BIT);                        /* reset Pixels            */
   glColor3f (1.0, 1.0, 1.0);                            /* Farbe weiss             */
   glPushMatrix();                                       /* sichere Matrix          */
   glRotatef(30.0, 0.0, 1.0, 0.0);                       /* drehe 30 Grad um y      */
   glRotatef(30.0, 1.0, 0.0, 0.0);                       /* drehe 30 Grad um x      */
   glutWireSphere(1.0, 20, 16);                          /* zeichne die Sonne       */
   glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);            /* Drehung um Sonne        */
   glTranslatef (2.0, 0.0, 0.0);                         /* Verschiebung von Sonne  */
   glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);             /* Erd-Drehung             */
   glutWireSphere(0.2, 10, 8);                           /* zeichne die Erde        */
   glPopMatrix();                                        /* restauriere Matrix      */
   glutSwapBuffers();                                    /* tausche Puffer          */
}

void keyboard (unsigned char key, int x, int y){
   switch (key) {                                        /* abhaengig von der Taste */
     case 'd': day =(day+10) % 360; break;               /* erhoehe Tag um 10       */
     case 'y': year=(year+5) % 360; break;}              /* erhoehe Jahr um 5       */
   glutPostRedisplay();                                  /* rufe display auf        */
}

int main (int argc, char ** argv) {          /* Hauptprogramm                        */
  glutInit(&argc, argv);                     /* initialisiere GLUT                   */
  glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB); /* double buffer, true color            */
  glutInitWindowSize(750, 500);              /* initiale Fenstergroesse              */
  glutInitWindowPosition(100,100);           /* initiale Fensterposition             */
  glutCreateWindow("planet");                /* Fenster mit Aufschrift               */
  init();                                    /* rufe init auf                        */
  glutDisplayFunc(display);                  /* registriere display                  */
  glutKeyboardFunc(keyboard);                /* registriere keyboard                 */
  glutMainLoop();                            /* beginne Event-Schleife               */
  return 0;                                  /* ISO C verlangt Rueckgabe             */
}
```

planet.c: Interaktion durch Tastatur zur Rotationsbewegung

```
#include <GL/glut.h>                          /* Header fuer OpenGl utility toolkit */
#include <stdlib.h>                           /* header fuer C-library              */

void init(void) {
   glClearColor (0.0, 0.0, 0.0, 0.0);         /* Hintergrundfarbe schwarz           */
   glShadeModel (GL_SMOOTH);                  /* smooth shading                     */
}

void display(void) {
   glClear (GL_COLOR_BUFFER_BIT);             /* reset Farbpuffer                   */
   glBegin (GL_TRIANGLES);                    /* Beginn der Dreiecks-Knoten         */
   glColor3f  ( 1.0,  0.0, 0.0);              /* Farbe rot                          */
   glVertex2f (25.0,  5.0);                   /* Knoten unten rechts                */
   glColor3f  ( 0.0,  1.0, 0.0);              /* Farbe gruen                        */
   glVertex2f ( 5.0, 25.0);                   /* Knoten links oben                  */
   glColor3f  ( 0.0,  0.0, 1.0);              /* Farbe blau                         */
   glVertex2f ( 5.0,  5.0);                   /* Knoten unten links                 */
   glEnd();                                   /* Ende der Dreiecks-Knoten           */
   glFlush ();                                /* direkt ausgeben                    */
}

void reshape (int w, int h) {
   GLfloat p = (GLfloat) w / (GLfloat) h;     /* berechne Fensterverhaeltnis        */
   glViewport (0,0,(GLsizei)w,(GLsizei)h);    /* lege Viewport fest                 */
   glMatrixMode (GL_PROJECTION);              /* ab jetzt Projektionsmatrix         */
   glLoadIdentity ();                         /* lade Einheitsmatrix                */
   if (p > 1.0)                               /* falls breiter als hoch             */
        gluOrtho2D (0.0,p*30.0,0.0,30.0  );   /* left,right,bottom,top              */
   else gluOrtho2D (0.0,  30.0,0.0,30.0/p);   /* left,right,bottom,top              */
   glMatrixMode(GL_MODELVIEW);
}

int main (int argc, char ** argv) {           /* Hauptprogramm                      */
  glutInit(&argc, argv);                      /* initialisiere GLUT                 */
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);  /* single buffer, true color          */
  glutInitWindowSize(750, 500);               /* initiale Fenstergroesse            */
  glutInitWindowPosition(100,100);            /* initiale Fensterposition           */
  glutCreateWindow("smooth");                 /* Fenster mit Aufschrift             */
  init();                                     /* rufe init auf                      */
  glutDisplayFunc(display);                   /* registriere display                */
  glutReshapeFunc(reshape);                   /* registriere reshape                */
  glutMainLoop();                             /* beginne Event-Schleife             */
  return 0;                                   /* ISO C verlangt Rueckgabe           */
}
```

smooth.c: Dreieck mit Farbverlauf durch Interpolation

```
#include <GL/glut.h>                                    /* fuer OpenGl utility toolkit*/
#include <stdlib.h>                                     /* fuer C-Library             */

GLuint listName;                                        /* Handle fuer Display-Liste  */

void init (void) {
   listName = glGenLists (1);                           /* trage Namen ein            */
   glNewList (listName, GL_COMPILE);                    /* erzeuge Display-Liste      */
      glColor3f (1.0, 0.0, 0.0);                        /* Farbe rot                  */
      glBegin (GL_TRIANGLES);                           /* Beginn Dreiecks-Punkte     */
      glVertex2f (0.0, 0.0);                            /* Knoten links unten         */
      glVertex2f (1.0, 0.0);                            /* Knoten rechts unten        */
      glVertex2f (0.0, 1.0);                            /* Knoten links oben          */
      glEnd ();                                         /* Ende der Dreiecksliste     */
      glTranslatef (1.5, 0.0, 0.0);                     /* Verschiebe nach rechts     */
   glEndList ();                                        /* Ende der Display-Liste     */
   glShadeModel (GL_FLAT);                              /* verwende Flat-Shading      */
}

void display(void) {                                    /* Anzeige-Routine            */
   GLuint i;                                            /* Integer-Variable           */
   glClear (GL_COLOR_BUFFER_BIT);                       /* reset Farb-Puffer          */
   for (i=0; i<10; i++) glCallList(listName);           /* rufe Display-Liste auf     */
   glBegin (GL_LINES);                                  /* Beginn Linien-Punkte       */
   glVertex2f (0.0,0.5); glVertex2f(15.0,0.5);          /* zwei Knoten                */
   glEnd ();                                            /* Ende Linen-Punkte          */
   glFlush ();                                          /* direkt ausgeben            */
}

void reshape(int w, int h) {                            /* Reshape-Routine            */
   GLfloat p = (GLfloat) w / (GLfloat) h;               /* Fensterverhaeltnis         */
   glViewport(0, 0, w, h);                              /* setze Viewport             */
   glMatrixMode(GL_PROJECTION); glLoadIdentity();       /* ab jetzt Projektion        */
   if (p > 1 ) gluOrtho2D (0.0, 2.0*p, -0.5, 1.5);      /* links, rechts, unten, oben */
         else  gluOrtho2D (0.0, 2.0, -0.5/p, 1.5/p);    /* links, rechts, unten, oben */
   glMatrixMode(GL_MODELVIEW); glLoadIdentity();        /* ab jetzt Modelview         */
}

int main (int argc, char ** argv) {                     /* Hauptprogramm              */
  glutInit(&argc, argv);                                /* initialisiere GLUT         */
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);            /* single buffer, true color  */
  glutInitWindowSize(800, 100);                         /* initiale Fenstergroesse    */
  glutCreateWindow("list");                             /* Fenster mit Aufschrift     */
  init();                                               /* rufe init auf              */
  glutDisplayFunc(display);                             /* registriere display        */
  glutReshapeFunc(reshape);                             /* registriere reshape        */
  glutMainLoop();                                       /* beginne Event-Schleife     */
  return 0;                                             /* ISO C verlangt Rueckgabe   */
}
```

list.c : Dreiecke definiert durch Display-Liste

```
#include <GL/glut.h>
#include <stdlib.h>

GLfloat ctrlpoints[4][3] = {
        {-4.0, -4.0, 0.0}, {-2.0, 4.0, 0.0},
        { 2.0, -4.0, 0.0}, { 4.0, 4.0, 0.0}};

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
   glMatrixMode(GL_PROJECTION); glLoadIdentity();
   glOrtho(-7.5, 7.5, -5.0, 5.0, -5.0, 5.0);
   glMatrixMode(GL_MODELVIEW); glLoadIdentity();
}

void display(void)
{
   int i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINE_STRIP);
      for (i=0; i<=30; i++) glEvalCoord1f((GLfloat) i/30.0);
   glEnd();
   glPointSize(5.0);
   glColor3f(1.0, 1.0, 0.0);
   glBegin(GL_POINTS);
      for (i=0; i<4; i++) glVertex3fv(&ctrlpoints[i][0]);
   glEnd();
   glFlush();
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (750, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("bezier-curve");
   init ();
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

bezier-curve.c: zweidimensionale Bezier-Kurve

```
#include <stdlib.h>
#include <GL/glut.h>

GLfloat ctrlpoints[4][4][3] = {
   {{-1.5, -1.5,  4.0}, {-0.5, -1.5,  2.0}, {0.5, -1.5, -1.0}, {1.5, -1.5,  2.0}},
   {{-1.5, -0.5,  1.0}, {-0.5, -0.5,  3.0}, {0.5, -0.5,  0.0}, {1.5, -0.5, -1.0}},
   {{-1.5,  0.5,  4.0}, {-0.5,  0.5,  0.0}, {0.5,  0.5,  3.0}, {1.5,  0.5,  4.0}},
   {{-1.5,  1.5, -2.0}, {-0.5,  1.5, -2.0}, {0.5,  1.5,  0.0}, {1.5,  1.5, -1.0}}
};

void display(void) {
   int i, j;
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glPushMatrix ();
   glRotatef(85.0, 1.0, 1.0, 1.0);
   for (j = 0; j <= 8; j++) {
      glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++) glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
      glEnd();
      glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++) glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
      glEnd();
   }
   glPopMatrix ();
   glFlush();
}

void init(void) {
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &ctrlpoints[0][0][0]);
   glEnable(GL_MAP2_VERTEX_3);
   glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
   glMatrixMode(GL_PROJECTION); glLoadIdentity();
   glOrtho(-6.0, 6.0, -4.0, 4.0, -4.0, 4.0);
   glMatrixMode(GL_MODELVIEW); glLoadIdentity();
}

int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize (750, 500); glutInitWindowPosition (100, 100);
   glutCreateWindow ("bezier-surface"); init ();
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

bezier-surface.c : dreidimensionale Bezierkurve

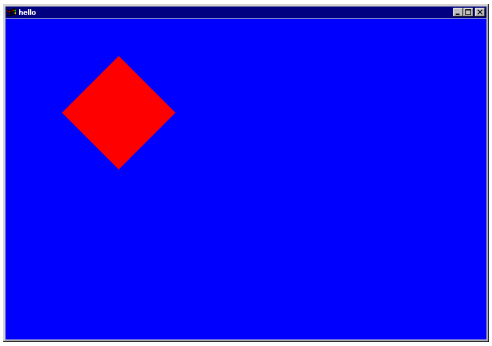
```
#include <GL/glut.h>
#include <stdlib.h>

void init(void) {
   GLfloat mat_diffuse[]      = { 1.0, 0.3, 0.3, 1.0 };
   GLfloat mat_specular[]     = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[]    = { 50.0 };
   GLfloat light_position[]   = { 1.0, 1.0, 1.0, 0.0 };
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glMaterialfv (GL_FRONT,  GL_DIFFUSE,   mat_diffuse);
   glMaterialfv (GL_FRONT,  GL_SPECULAR,  mat_specular);
   glMaterialfv (GL_FRONT,  GL_SHININESS, mat_shininess);
   glLightfv    (GL_LIGHT0, GL_POSITION,  light_position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}

void display(void) {
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glRotatef(-40.0, 0.0, 1.0, 0.0);
   glRotatef( 20.0, 1.0, 0.0, 0.0);
   glutSolidTeapot(1.0);
   glPopMatrix(); glFlush ();
}

void reshape (int w, int h){
   GLfloat p = (GLfloat) w / (GLfloat) h;
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION); glLoadIdentity();
   if (p > 1.0) glOrtho(-1.5*p,1.5*p,-1.5,1.5,-10.0,10.0);
           else glOrtho(-1.5,1.5,-1.5/p,1.5/p,-10.0,10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
   glutInitWindowSize (750, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("teapot");
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop(); return 0;
}
```

teapot.c: Teekanne mit Beleuchtung

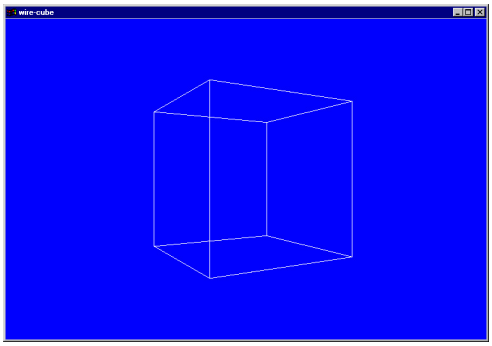
```
#include <GL/glut.h>                                        /* OpenGl Utiltiy Toolkit  */
#include <stdlib.h>                                         /* C-library               */

int g;                                                      /* Variable fuer Gradzahl  */

void init(void) {
   GLfloat mat_diffuse[]      = { 1.0, 0.3, 0.3, 1.0 };  /* diffuse Farbe           */
   GLfloat mat_specular[]     = { 1.0, 1.0, 1.0, 1.0 };  /* spekulare Farbe         */
   GLfloat mat_shininess[]    = { 50.0 };                 /* Reflexionskoeffizient   */
   GLfloat light_position[]   = { 1.0, 1.0, 1.0, 0.0 };  /* Lichtquellenposition    */
   glClearColor (0.0, 0.0, 0.0, 0.0);                      /* Hintergrundfarbe        */
   glShadeModel (GL_SMOOTH);                               /* smooth shading          */
   glMaterialfv(GL_FRONT, GL_DIFFUSE,  mat_diffuse);       /* diffuse Farbzuweisung   */
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);      /* spekulare Farbzuweisung */
   glMaterialfv(GL_FRONT, GL_SHININESS,mat_shininess);     /* Reflexionszuweisung     */
   glLightfv   (GL_LIGHT0,GL_POSITION, light_position);    /* Lichtpositionszuweisung */
   glEnable(GL_LIGHTING); glEnable(GL_LIGHT0);             /* Beleuchtung aktivieren  */
   glEnable(GL_DEPTH_TEST);                                /* Tiefentest aktiveren    */
   glMatrixMode (GL_PROJECTION); glLoadIdentity();         /* ab jetzt Projektion     */
   glOrtho(-1.95,1.95,-1.5,1.5,-10.0,10.0);                /* links,rechts,unten,oben */
   glMatrixMode(GL_MODELVIEW); glLoadIdentity();           /* ab jetzt Modelview      */
}

void display(void) {
   glClear (GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);      /* reset Buffer            */
   glPushMatrix();                                         /* rette Matrix            */
   glRotatef((GLfloat) g, 0.0, 1.0, 0.0);                  /* g grad Grad um y-Achse  */
   glRotatef( 20.0, 1.0, 0.0, 0.0);                        /* 20 Grad um x-Achse      */
   glutSolidTeapot(1);                                     /* zeichne Teapot          */
   glPopMatrix();                                          /* restauriere Matrix      */
   glutSwapBuffers();                                      /* wechsele Puffer         */
}

void spinDisplay(void) {
  g = (g + 5) % 360;                                       /* Gradzahl um 5 erhoehen  */
  glutPostRedisplay();                                     /* display erneut aufrufen */
}

int main(int argc, char** argv) {
   glutInit(&argc, argv);                                  /* initialisiere GLUT      */
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH); /* Double buffer, ...      */
   glutInitWindowSize (750, 500);                          /* initiale Fenstergroesse */
   glutInitWindowPosition (100, 100);                      /* initiale Fensterposition*/
   glutCreateWindow ("teapot-rotate");                     /* Fenster mit Aufschrift  */
   init ();                                                /* Initialisierung         */
   glutDisplayFunc(display);                               /* registriere display     */
   glutIdleFunc(spinDisplay);                              /* registriere spinDisplay */
   glutMainLoop(); return(0);                              /* starte Hauptschleife    */
}
```

teapot-rotate.c: rotierende Teekanne

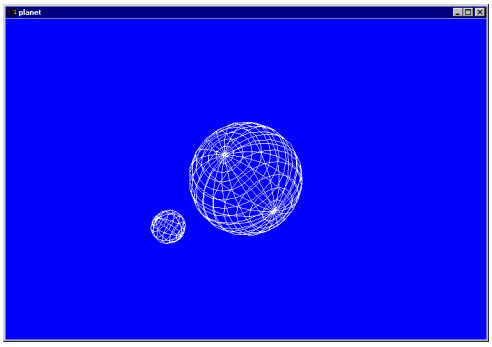
24.4 Screenshots



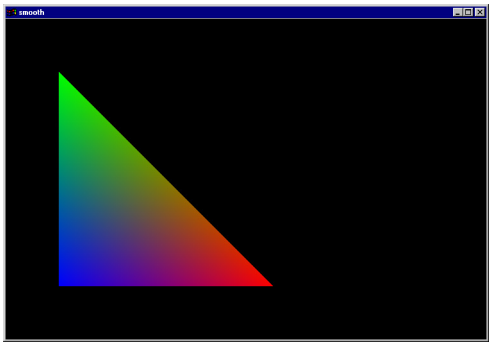
hello.exe



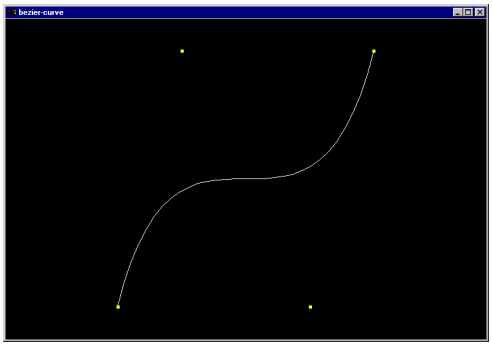
wire-cube.exe



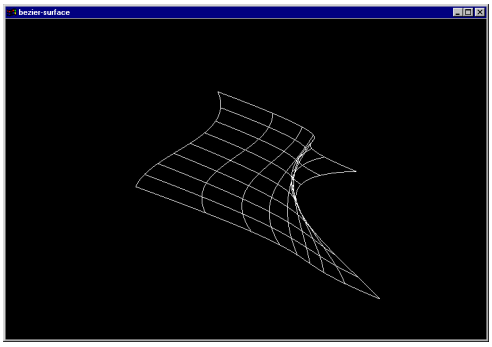
planet.exe



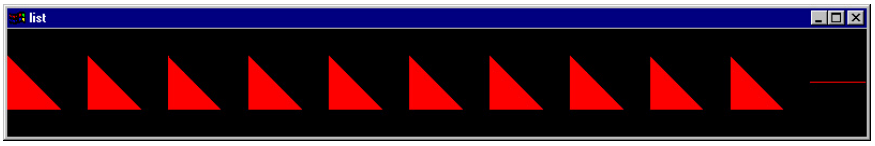
smooth.exe



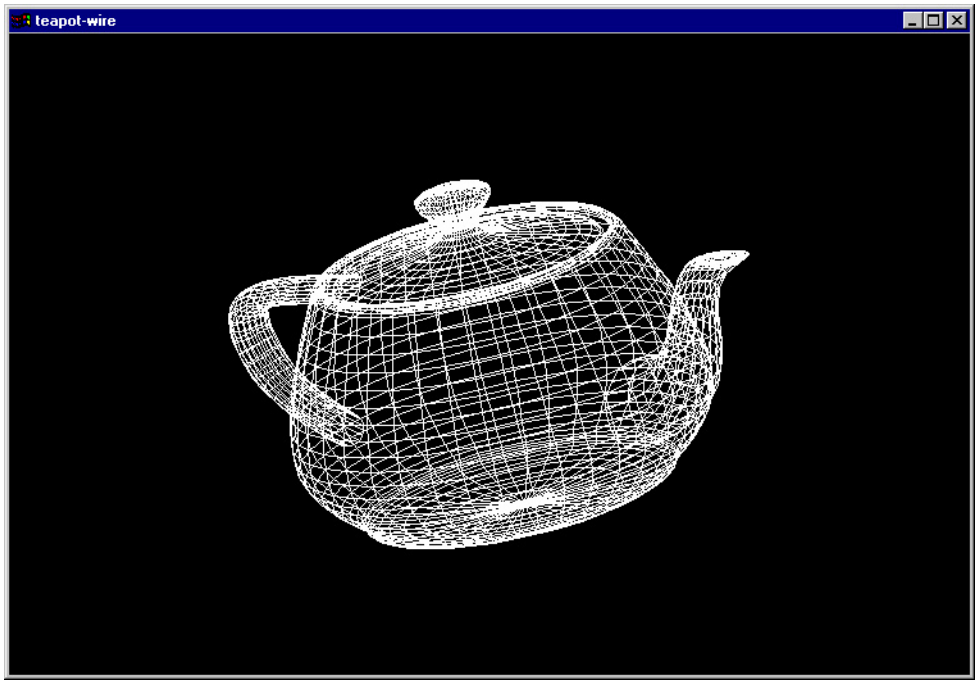
bezier-curve.exe



bezier-surface.exe



list.exe



Drahtgitter für teapot

teapot.exe