Kapitel 13
Computergrafik

13.1 Einleitung

Wer Programme mit Computergrafik entwickeln möchte, kommt vermutlich am Anfang kaum an OpenGL vorbei, auch wenn er später vielleicht auf Werkzeuge wie „Visualisation Toolkit“ oder andere zurückgreift.[1] Die Beschäftigung mit Computergrafik besteht zunächst aus der Einarbeitung in die Theorie, was aufgrund der Vielzahl an Modellen und Komponenten, die in der Endgrafik berücksichtigt worden sind, schon ein gehöriges Stück Arbeit darstellt.

Mit der Theorie werden wir uns hier nur am Rande beschäftigen, da diese in vielen Lehrbüchern ausführlich beschrieben ist. Parallel zur Theorie ist eine praktische Arbeit recht nützlich, da die Anschauung und Übung vieles erklärt, was in der Theorie etwas nebulös geblieben ist. OpenGL ermöglicht es, sehr schnell einfache Ergebnisse auf dem Bildschirm zu sehen und schrittweise zu komplexeren Themen aufzusteigen.

Zum Systemverständnis ist zunächst zu bemerken, dass in modernen Rechnern weitgehende Arbeitsteilung zwischen den verschiedenen Komponenten herrscht. Die CPU ist in der Regel dafür zuständig, Koordinaten, Rechenwerte und Parameter auf die Grafikkarte zu laden und dabei Tastatur-, Maus- oder sonstige Ereignisse zu berücksichtigen. Der Prozessor auf der Grafikkarte übernimmt die recht aufwändigen Berechnungen der tatsächlich darzustellenden Daten bis hin zu low-level-Operationen und ist für diese Aufgaben spezialisiert. In der Literatur werden auch diese Operationen ausführlich mit allen mathematischen Details vorgestellt, doch muss man schon eine ganze Menge Aufwand treiben, bis man selbst größere Rechenoperationen in der eigenen Programmierung vornehmen muss. Wir werden beim Entwurf der Softwarekomponenten diese Arbeitsteilung berücksichtigen und möglichst viel dem Rechenprozessor zu überlassen.

[1] Was man verwendet oder besser verwenden kann, hängt stark vom Anwendungsbereich ab. Will man Spiele programmieren? Oder technisch-medizinische Visualisierungen produzieren? Oder fotorealistische künstliche Bilder berechnen? Je nach Bereich existieren unterschiedliche Methoden und Werkzeuge, so dass hier kein allgemeiner Rat gegeben werden kann.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_13, © Springer-Verlag Berlin Heidelberg 2010

Diese Arbeitsteilung machen die OpenGL-Aufrufe stark C-lastig, so dass die objektorientierte Programmierung etwas auf der Strecke bleibt. Entsprechend sehen viele Tutorien mit Praxisbeispielen aus. Den Mangel wollen wir hier beheben, wobei einige C++ - Spezifika wie mehrfache Vererbung zum Einsatz kommen. Das Framework ist primär für Lehrzwecke konzipiert – falls sich daraus mehr machen lässt, freut mich das zwar, aber das muss jeder selbst ausloten.

Zur Systemvorbereitung laden Sie die OpenGL-Bibliothek sowie die GLU- und die GLUT-Bibliotheken oder öffnen Sie einfach ein OpenGL-Projekt in Ihrer Entwicklungsumgebung.

13.2 Systemumgebung

13.2.1 Systeminitialisierung

Ein OpenGL-Grafikprogramm sieht zunächst aus wie ein normales C/C++ - Programm. An Systemkomponenten werden die gl-, die glu- und die glut-Bibliothek benötigt, wobei die letzten beiden praktische Hilfsmethoden für den einfachen Umgang mit der gl-Bibliothek und dem Betriebssystem bereitstellen.[2] Eine der ersten Anweisungszeilen muss das OpenGL-System aktivieren. Der Systembefehl hierzu lautet

```
glutInit(&argc,argv);
```

wobei die Standardparameter der `main`-Methode als Parameter zu übergeben sind. Im Weiteren werden die Grafikobjekte und das Grafikfenster vorbereitet. Wenn die Grafik schließlich angezeigt werden soll, muss die Kontrolle an die OpenGL-Laufzeitumgebung übergeben werden. Deren Aufruf (`glutMainLoop()`) ist dann der letzte Befehl im Hauptprogramm, denn aus dieser Umgebung gibt es keinen Rücksprung mehr. Ein Schließen der Grafik bedeutet auch das Ende des Programms.

Um dem Anwender Gelegenheit zu geben, andere Aufgaben zu erledigen bzw. auf Ereignisse wie Tastatureingaben oder Mausbewegungen zu reagieren, können eine Reihe von Ereignisbehandlungsmethoden definiert und dem Laufzeitsystem vor der Kontrollübergabe mitgeteilt werden. Tritt ein Ereignis ein, beispielsweise die Änderung der Fenstergröße des Grafikfensters, wird die angegebene Methode vom OpenGL-System aufgerufen und ausgeführt. Die Methoden sollten die Kontrolle schnellstmöglich wieder an das OpenGL-System durch Rücksprung übergeben, denn ein Arbeiten mit mehreren Threads ist meist weder vorgesehen noch möglich und das Grafiksystem ist so lange blockiert, wie die Kontrolle in den Methoden bleibt.

---

[2] Um Überraschungen mit dem Linker/Compiler in einigen Systemen zu vermeiden, sollten alle drei Header in dieser Reihenfolge eingebunden werden.

13.2.2 System-Basisklasse und aktives Objekt

Als zentrale Schnittstelle zwischen OpenGL-System und dem Anwendungsprogramm definieren wir die Klasse `GrafikSystem`:

```
class GrafikSystem {
public:
    typedef void(*func)();

    GrafikSystem();
    virtual ~GrafikSystem();
```

Eine Grafikszene ist in der Regel mit einem kompletten Grafikobjekt verbunden. Um zwischen Szenen problemlos wechseln zu können, sehen wir den Aufbau mehrerer Grafikobjekte vor, von denen eines aktiv ist und die anderen im aktiven registriert werden. Zum Szenenwechsel kann zwischen den Objekten umgeschaltet werden.[3]

```
static GrafikSystem* get_active_object();
static GrafikSystem* swap_active_object(
                              GrafikSystem* obj);

static void register_timed_object(
                    int msec, GrafikSystem* obj);
static void unregister_timed_object(
                              GrafikSystem* obj);

static int register_object(GrafikSystem* obj);
static void unregister_object(int handle);
static GrafikSystem* get_registered_onject(
                                    int handle);
```

OpenGL erlaubt es dem Anwender, eine Reihe von so genannten CallBack-Funktionen zu registrieren. Diese legen wir weiter unten im `private`-Teil als statische Funktionen an, da feste Funktionsadressen übergeben werden müssen. Die statischen Funktionen haben mehr oder weniger nur die Aufgabe, auf die objektspezifischen CallBack-Funktionen umzuschalten, die wir hier als virtuelle Methoden einführen.

```
virtual void init();
virtual void display();
virtual void reshape(int,int);
virtual void keyboard(unsigned char,int,int);
virtual void keyboard_func_keys(int,int,int);
virtual void idle();
```

[3] Alternativ kann man natürlich auch innerhalb eines Objektes zwischen dargestellten und nicht dargestellten Grafikobjekten wechseln. Die Kapselung aktiver Grafikobjekte in einem aktiven Systemobjekt hat jedoch einige Vorteile.

```
virtual void mouse(int,int,int,int);
virtual void mouse_motion(int,int);
virtual void timer();
```

Die folgenden Methoden stellen die Grafikszene dar. Auf die Details gehen wir später ein.

```
void swap_perspective_view(bool view);
bool show_grafic();
```

Die folgenden öffentlichen Attribute definieren unter anderem die Fenstergröße auf dem Bildschirm, das Raumelement, innerhalb dessen Objektkoordinaten liegen müssen, damit das Objekt dargestellt wird, Kameraeigenschaften wie Position, Blickrichtung und Öffnungswinkel bei perspektivischer Darstellung und weiteres. Der `private`-Teil enthält schließlich die bereits diskutierten statischen CallBack-Methoden.

```
    unsigned int disp_mode;
    unsigned int clear_mode;
    bool display_init;

    struct Window {
        int width;
        int height;
        int pos_x;
        int pos_y;
    } window;

    Color bg_color, standard_color;
    GLfloat depth_value;

    struct VisibleVolume {
        double xmin, xmax,
               ymin, ymax,
               zmin, zmax;
    } volume;

    bool reshape_window;
    bool reshape_volume;

    bool perspective_view;

    struct Camera {
        Point location;
        Point direction;
        Point up;
    } camera;

    struct Lens {
        GLfloat angle;
```

```
        GLfloat yx;
        GLfloat zmin,zmax;
    } lens;

private:
    void initi();
    static void disp(void);
    static void resh(int,int);
    static void keyb(unsigned char,int,int);
    static void keybfc(int,int,int);
    static void idl(void);
    static void maus(int,int,int,int);
    static void mausmotion(int,int);
    static void tim(int);
};//end class
```

Fehler- und Plausibilitätsprüfungen sind in der Regel nicht vorhanden, um keine Geschwindigkeitseinbußen in Kauf nehmen zu müssen. Zur Vorbereitung einer Grafik können in der `main`-Funktion beliebige Objekte von Klassen, die von der Systemklasse erben, erzeugt und in Containern montiert werden. Um eine Grafik anzuzeigen, ist von einem dieser Objekte die Methode `show_grafic()` aufzurufen, die bei Erfolg das Grafiksystem startet und nicht mehr zurückkehrt. Die Methode führt folgende Aktionen aus:

```
bool GrafikSystem::show_grafic(){
    active_object=this;
    glutInitDisplayMode (disp_mode);
    glutInitWindowSize (window.width,window.height);
    glutInitWindowPosition (window.pos_x,window.pos_y);
    glutCreateWindow ("Grafikobjekt");
    initi();
    glutDisplayFunc(disp);
    ...
    glutMainLoop();
}//end function
```

Zunächst wird hier ein aktives Objekt definiert. Hierbei handelt es sich um eine statische Zeigervariable innerhalb des Definitionsmoduls der Klasse, auf der ein Zeiger auf das Objekt hinterlegt wird, von dem die Grafik gezeichnet wird. Eine solche Hinterlegung ist notwendig, da die Bibliothek objektorientiert ist, die OpenGL-Aufrufmethoden aber reine C-Methoden sind. In den von OpenGL aufgerufenen statischen Methoden kann über diese Zeigervariable auf das Objekt nebst seinen Attributen zurückgegriffen werden.

Statische Methoden in der Systemklasse erlauben auch einen Zugriff auf das aktive Objekt bzw. den Austausch gegen ein anderes Objekt.

```
static GrafikSystem* active_object=0;

GrafikSystem*
GrafikSystem::swap_active_object(GrafikSystem* obj){
    GrafikSystem* tmp=active_object;
    active_object=obj;
    active_object->initi();
    active_object->reshape(tmp->window.width,
                           tmp->window.height);
    return tmp;
}
```

Für die Verwaltung der aller erzeugten Objekte wird ein einfaches Registrierungssystem implementiert, das die Objekte in einem Container speichert.

```
static map<int,GrafikSystem*> timed_objects;
static map<int,GrafikSystem*> registered_objects;
```

Aufgabe. Derartige Registrierungssysteme haben wir bereits mehrfach verwendet, so dass diese Implementation komplett Ihnen überlassen sei.

Die weiteren Methodenaufrufe in `show_grafic()` definieren das Grafikfenster, genauer den Arbeitsmodus sowie die Position und Größe des Grafikfensters. Die Fenstergröße wird im Weiteren benötigt, um die Grafikobjekte unverzerrt darstellen zu können, und wird daher in speziellen Attributen in der Klasse gespeichert und im Konstruktor mit sinnvollen Werten initialisiert. Außerdem erfolgt die Übergabe der Adressen der Ereignisbehandlungsmethoden und abschließend dem Aufruf des OpenGL-Laufzeitsystems.

13.2.3 Objektinitialisierung und Projektionsmatrizen

In der Liste der CallBack-Methoden taucht die Methode `init()` auf, die eigentlich keine CallBack-Methode ist. Jede Klasse in der Klassenhierarchie kann eine Initialisierungsmethode `init()` definieren, deren leerer virtueller Prototyp in der Basisklasse definiert ist. Die Initialisierungsmethode wird einmalig zu Beginn aufgerufen, typischerweise vor dem Öffnen des Grafikfensters, um eine Szene für die Darstellung vorzubereiten. Eine Szene kann typischerweise mehrere Objekte beinhalten, die in einem Szenenobjekt montiert werden. Eine Reihe von Darstellungsparametern ist unter Umständen erst dann ermittelbar, wenn alle Teilobjekte montiert sind, und diese Aufgaben werden durch die `init()`-Methode übernommen.

Beim Studium der `show_grafic()`-Methode fällt auf, dass nicht `init()`, sondern `initi()` aufgerufen wird. Hierbei handelt es sich nicht um einen Tippfehler, sondern die aufgerufene Methode gehört zu den privaten Methoden und erledigt noch einige zusätzliche Aufgaben, die im Rahmen jeder Initialisierung auftreten:

```
void GrafikSystem::initi(){
    init();
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(volume.xmin,volume.xmax, volume.ymin,
            volume.ymax, volume.zmin, volume.zmax);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}//end function
```

Zu diesen Aufgaben gehört die Initialisierung des sichtbaren Bereiches, der hier nicht-perspektivisch initialisiert wird,[4] sowie des Tiefentests, der für die korrekte Überdeckung voreinander liegender Flächen sorgt. Hier sind einige Erklärungen über die Art und Weise, wie OpenGL Objekte darstellt, angebracht.

Grafische Objekte sind in der Regel 3-dimensional und werden vom Grafiksystem auf die 2-dimensionale Bildschirmebene projiziert. Um feststellen zu können, welches von mehreren Objekten, die gleiche Koordinaten auf dem Bildschirm besitzen, dargestellt werden soll, wird die dritte Koordinate ausgewertet. Was näher an der Projektionsebene liegt wird dargestellt. Die Prüfung heißt Tiefentest und muss aktiviert werden.

Wie weiter unten noch ausführlicher erläutert wird, lassen sich alle Verschiebungen, Drehungen und sonstige Manipulationen von Objekten mit Hilfe der Matrizenalgebra (→ lineare Algebra) durchführen. Einzelne Operationen können in Form von Matrizen dargestellt werden, und um mehrere Operationen hintereinander auszuführen, werden die dazu gehörenden Matrizen miteinander multipliziert. Die Operationen und die damit verbundenen Matrizen lassen sich in zwei Klassen unterteilen:

(a) GL_MODELVIEW umfasst alle Manipulationen der Objekte im absoluten 3-dimensionalen Koordinatensystem und ist unabhängig davon, welche Objekte im Fenster tatsächlich sichtbar sind.
(b) GL_PROJECTION umfasst die Projektion der Objekte auf die Sichtebene (den Bildschirm) an der Position des Betrachters (Kamera) und schneidet alle nicht sichtbaren Objekte aus.

Die Matrizen für beide Operationsklassen sind in OpenGL getrennt zu laden. Ladeoperationen für Matrizen können mehrere hintereinander auszuführenden Operationen umfassen, weshalb zu Beginn die zu ladende Matrix durch `glLoadIdentity()` als Einheitsmatrix initialisiert wird.

Die einfachste Projektion ist die Parallelprojektion. Vereinbarungsgemäß ist die Projektionsebene eine zur XY-Ebene parallele Ebene in der Entfernung Z vom Nullpunkt, die Projektionsstrahlen sind zur Z-Achse parallele Strahlen.

[4] Die perspektivische Sicht behandeln wir später

Die Blickrichtung ist folglich die negative Z-Achse. Die Projektion besteht aus dem Streichen der Z-Komponente eines Objektes, wobei das OpenGL-System Überdeckungen berücksichtigt, d.h. bei Objekten mit überdeckenden XY-Koordinaten nur das Objekt mit größerer Z-Koordinate zeichnet. Ein Drahtmodell eines Würfels mit einer Frontalansicht einer Fläche wird somit als Quadrat in der Farbe der vorderen Kanten gezeichnet.

Die Methode `glOrtho(..)` legt das sichtbare Volumen fest – alles außerhalb des definierten Quaders wird grundsätzlich ausgeschnitten. Die Quaderkoordinaten werden in Attributen hinterlegt und im Konstruktor initialisiert.

Abschließend wird GL_MODELVIEW aktiviert, da in den weiteren Operationen in der Regel nur noch Transformationsmatrizen bearbeitet werden.[5]

HINWEIS! Bei Wechsel des aktiven Objekts wird die `initi()`-Methode sowie die `reshape()`-Methode für das neue aktive Objekt aufgerufen. Hierdurch werden automatisch verschiedene Einstellungen für die neue Szene aktiviert. Eine Übergabe weiterer Parameter muss durch die Anwendung selbst erfolgen. Sofern in der Initialisierungsmethode aufwändige nur einmalig notwendige Berechnungen durchgeführt werden, sollte die Anwendung eine logische Steuerung zur Unterdrückung der erneuten Berechnung bei Reaktivierung aufweisen.

13.2.4 Ereignisfunktionen

Ereignisse wie Tastatureingaben, Mausbewegungen und Maustastenfunktionen, Timerereignisse und anderes werden von OpenGL durch den Aufruf von Ereignismethoden behandelt. In diesem Methoden können beliebige Aktionen wie Dynamisierung oder Austausch der Szene vorgenommen werden. Die Adressen der Funktionen wird dem OpenGL durch Anweisungen des Typs

```
glutReshapeFunc(resh);
```

festgelegt.[6] Wie der Klassendefinition zu entnehmen ist, werden hier die Adressen privater statischer Klassenmethoden übergeben, die für die Weitergabe an die zuständige Objektmethode zuständig sind:

```
void GrafikSystem::idl(void) {active_object->idle();}
```

Die Objektmethoden sind als virtuelle Methoden in der Basisklasse angelegt. Die meisten Methoden sind erst für konkrete Objekte mit Leben zu erfüllen, so dass wir hier nicht weiter darauf eingehen.

---

[5] Ausnahmen sind beispielsweise Zoom-Operationen, in denen die Projektionsmatrix angepasst wird.

[6] Es existiert eine so große Zahl solcher Methoden, dass man beim ersten Betrachten vermutlich erstaunt ist, wie viele Arten möglicher Ereignisse man übersehen hat. Wir beschränken hier die Vorstellung auf wenige Beispiele; weiteres entnehme der Leser dem GLUT-Handbuch. Im Projekt sind nicht alle Methoden implementiert; dem Leser wird es jedoch nicht schwer fallen, anhand der Systematik weitere Methoden zu installieren.

Zu beachten ist allerdings, dass für die Darstellung der Objekte die `display()`-Methode zuständig ist, die jedoch vom System nur dann aufgerufen wird, wenn für dieses klar ist, dass sich das darzustellende Objekt geändert hat. In den meisten CallBack-Methoden muss aber keine Änderung stattgefunden haben, so dass das GL-System die `display()`-Methode nicht aufruft und sich der Bildschirminhalt nicht ändert. Soll eine neue Darstellung erfolgen, müssen Sie dies in Ihrer CallBack-Methode durch den Aufruf

```
glutPostRedisplay();
```

erzwingen. Das OpenGL-System sieht in diesem Fall einen Aufruf der `display()`-Methode nach der Rückkehr aus der CallBack-Methode vor.

Einige der Ereignisbehandlungsmethoden erledigen allerdings wieder Aufgaben, die immer wieder auftreten und daher besser zentral erledigt werden, als sich darauf zu verlassen, dass der Programmierer ihre Implementation nicht vergisst. Diese sind:

13.2.4.1 Die display()-Methode, Darstellung der Objekte

Die zentrale Methode zur Darstellung eines Objektes ist die `display()`-Methode, die regelmäßig aufgerufen wird, wenn sich das Fenster oder das Objekt selbst verändert hat. Änderungen werden auf dem Bildschirm erst dann sichtbar, wenn diese Methode ausgeführt wird. Bei der Neuberechnung einer Grafik ist es in den meisten Fällen notwendig, die alte Grafik zu entfernen und das System auf bestimmte Standardwerte zurückzusetzen. Dies wird als allgemeine Dienstleistung in der `disp()`-Methode übernommen, sofern nicht objektspezifische Gründe dagegen stehen (Steuerattribut):

```
void GrafikSystem::disp(void){
  if(active_object->display_init){
    if(active_object->clear_mode & GL_COLOR_BUFFER_BIT)
      glClearColor(active_object->bg_color.r,
                   active_object->bg_color.g,
                   active_object->bg_color.b,
                   active_object->bg_color.a);
    if(active_object->clear_mode & GL_DEPTH_BUFFER_BIT)
      glClearDepth(active_object->depth_value);
    glClear(active_object->clear_mode);
    glColor4fv((GLfloat*)&active_object->standard_color);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
  }//endif
  active_object->display();
}//end function
```

Zu Löschen ist in der Regel der Bildschirminhalt sowie der Tiefentestwert für die Überdeckungsbestimmung, wobei zunächst Standardwerte festgelegt werden. Ebenfalls festgelegt wird eine Standardfarbe, die allerdings nur so lange gilt, bis in einer der `display()`-Methoden eine andere Farbe spezifiert wird. Die Farbe ist ein vierdimensionaler Wert, von dem aber in der Regel nur die RGB-Komponenten verwendet werden. Die Initialisierung erfolgt im Konstruktor mit meist gebräuchlichen Werten.

Außer den beiden hier beschriebenen zurückgesetzten Parametern existieren eine Reihe weiterer, deren Beschreibung man den OpenGL-Handbüchern entnehme. Gegebenenfalls ist der Code der Methode sowie die Anzahl der Attribute anzupassen, so dass nur die notwendigen Rücksetzbits spezifiziert werden müssen.

13.2.4.2 Die reshape()-Methode, Anpassen an das Fenster

Bei Veränderungen des Grafikfensters auf der Arbeitsoberfläche, beispielsweise durch Verschieben, Vergrößern oder Verkleinern, wird vom OpenGL-System die Methode `reshape(w,h)` aufgerufen. Die Methode kann auch aus einer anderen Ereignismethode heraus aufgerufen werden, wenn beispielsweise die Fenstergröße per Tastatureingabe vorgegeben wird. Die Änderung des Grafikfensters hat Auswirkungen auf die Darstellung der Objekte, und die hierfür zuständigen Parameter werden von der Basisklasse verwaltet.

```
void GrafikSystem::reshape(int w,int h){
    if(!reshape_window) {
        glutReshapeWindow(window.width,window.height);
        return;
    }//endif
    window.width=w;
    window.height=h;
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    if(!reshape_volume) return;
    glMatrixMode(GL_PROJECTION);
    if (w <= h)
        glOrtho(volume.xmin,volume.xmax,
                volume.ymin*(GLfloat)h/(GLfloat)w,
                volume.ymax*(GLfloat)h/(GLfloat)w,
                volume.zmin, volume.zmax);
    ...
}//end function
```

Die Aufgabe der `reshape()`-Methode besteht zunächst im Aufruf der Methode `glViewPort()`, die die Fenstergröße ändert. Wir haben dies hier konditionell gestaltet, so dass das Grafikobjekt sich auch weigern kann, solchem Ansinnen nachzukommen, wenn von der Anwendungsseite her die Fenstergröße unveränderbar vorgegeben wird.

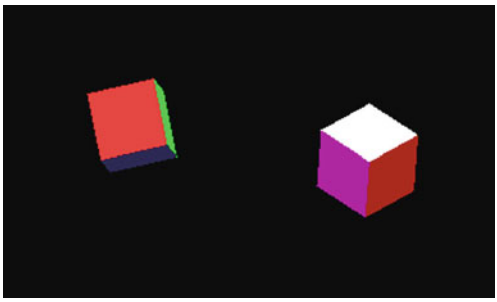


Abb. 13.1 Würfel in korrekter unverzerrter Darstellung

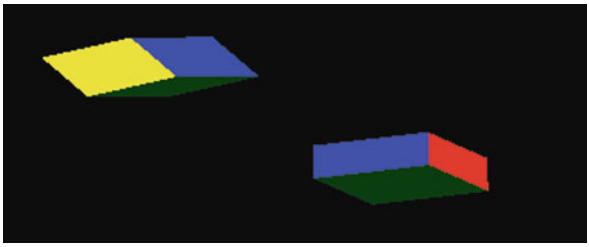


Abb. 13.2 Verzerrte Würfel nach Fensteränderung ohne Transformations-ausgleich

Nach Änderung der Fenstergröße kann/muss noch die Projektsmatrix angepasst werden. Das Attribut für das sichtbare Volumen wird dabei nicht verändert, die sichtbaren Koordinaten des Fensters jedoch auf das Verhältnis von Fensterbreite zu Fensterhöhe angepasst, so dass bei einer Verbreiterung Objektteile sichtbar werden können, die zuvor unsichtbar waren.

Hierdurch werden Objektverzerrungen vermieden. Die Wirkung fehlender Entzerrung ist in Abb. 13.2 gegenüber Abb. 13.1 mit Entzerrung demonstriert.

Die logischen Attribute `reshape_window` und `reshape_volume` erlauben die Fixierung oder Freigabe der Fenstergröße sowie die verzerrungsfreie oder verzerrte Anpassung der Inhalstdarstellung an die Fenstergröße. Die `reshape()`-Methode braucht in der Regel nicht von erbenden Klassen überschrieben werden.

13.2.4.3 Die idle()-Funktion, Dynamische Grafiken

Die `idle()`-Methode wird zyklisch vom OpenGL-System aufgerufen und erlaubt es, beliebige andere Aufgaben duchzuführen, beispielsweise komplette Dialoge mit dem Anwender abseits der `keyboard()`-Methode. Im Prinzip kann man beliebig lange in dieser Funktion verweilen, doch ist während dieser Zeit die weitere Grafikverarbeitung unterbrochen, so dass eine schnelle Rückkehr sinnvoll ist.

Die `idle()`-Methode wird häufig verwendet, um dynamische Grafiken zu erzeugen, indem beispielsweise Objektkoordinaten automatisch neu berechnet werden. Die Methode besitzt jedoch kein definiertes Zeitverhalten, sondern man kann nur die Anzahl der Aufrufe der Methode als Steuerparameter verwenden. Für

eine „schöne Dynamik“ ist jedoch ein sinnvolles Zeitverhalten Voraussetzung. Steuermöglichkeiten bestehen einerseits über den Aufruf von Systemuhrfunktionen oder wahlweise durch Nutzung der `glutTimer()`-callback-Funktion, die eine Vorgabe der Aufrufzyklen erlaubt.

WICHTIG! Werden bei den letzten Methoden Teile der Grafik verändert, so werden die Ergebnisse erst sichtbar, wenn die Grafik durch die `display()`-Methode neu gezeichnet wird. Das erfolgt jedoch NICHT standardmäßig, sondern muss durch den Befehl `glutPostRedisplay()` ausgelöst werden.

13.2.4.4 Die Timer-Funktion

OpenGL erlaubt die Registrierung von zeitgesteuerten Aufrufen von Methoden. Die registrierte Methode wird nach der angegebenen Anzahl von msec mit dem übergebenen Zahlenwert als Parameter aufgerufen, wobei mehrere verschiedene Zeitereignisse aktiviert werden können. Jedes Zeitereignis wird nur einmalig aufgerufen, so dass für zyklische Abläufe jeweils nach der Ausführung eine erneute Registrierung notwendig ist.

Die Registrierung von Objekten erfolgt durch statische Methoden in der Grafikklasse. Die Programmlogik entspricht weitgehend der Registration von Grafikobjekten, wie sie bereits bei der Diskussion des aktiven Objekts vorgestellt wurde.

13.3 Daten und Datencontainer

Wir stellen hier die einige Grundtypen von Datencontainern vor. Im Kapitel „Beleuchtungseffekte“ werden die Containertypen um weitere Spezifikationen ergänzt.

13.3.1 Punkte und Punktcontainer

Das Basisgrafiksystem von OpenGL arbeitet mit Punkten (auch als Ortsvektoren interpretierbar), die in verschiedener Weise in Punktgruppen organisiert und dann als Punkte, Linien oder Flächen dargestellt werden. Die Koordinaten von Punkten können in verschiedenen Datenformaten angegeben werden, wobei wir uns hier auf den Fließkommatyp `GLfloat` beschränken.

Punkte können nun mit zwei, drei oder vier Koordinaten angegeben werden. Mit diesen Koordinatenanzahlen hat es nun folgendes auf sich:

- Zweidimensionale Objekte verwendet man, wenn es sich um eine ebene Darstellung handelt, die auf den Bildschirm zu übertragen ist. Überdeckungen von Objekten muss man, falls vorhanden, in diesem Fall von Hand angeben.
- Dreidimensionale Objekte verwendet man, wenn sämtliche Operationen dem Grafiksystem überlassen werden.
- Vierdimensionale Objekte werden verwendet, wenn im Anwendungsprogramm Berechnungen vorgenommen werden sollen.

Berechnungen an Objekten bestehen meist aus Skalierungen, Drehungen und Verschiebungen. Drehungen im Raum werden durch Matrizen vermittelt, z.B. dreht

$$\vec{b} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \vec{a}$$

eine Drehung des Punktes $\vec{a}$ um die Z-Achse. Verschiebungen werden durch Vektoraddition vermittelt, d.h. Man benötigt zwei verschiedene Operationen, um Transformationen an Objekten auszuführen. Das ändert sich beim Übergang zu einer homogenen 4-dimensionalen Geometrie:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ t = 1 \end{pmatrix}$$

Wer ein wenig mit der Matrizenrechnung vertraut ist, verifiziert leicht, dass die Skalierung, Drehung und Verschiebung durch die Matrizen

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \; D = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \; T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

erzeugt werden, d.h. man kann mehrere Operationen hintereinander zu einer Transformationsmatrix vereinen (vergleiche auch die Anmerkungen zur Initialisierung der Matrizen in OpenGL).

Die homogene 4. Koordinate ist vereinbarungsgemäß immer Eins; bei perspektivischen Ansichten können auch andere Werte angenommen werden, wie wir noch sehen werden.[7]

Der Grafikprozessor selbst verwaltet und bearbeitet sämtliche Daten im homogenen 4-dimensionalen Koordinatensystem, weil eine einzige Matrix für die Ausführung komplexer Gesamttransformationen ausreicht. Für alle Format sind in der OpenGL-Bibliothek Schnittstellenfunktionen vorhanden, die die Koordinaten einzeln oder als Adressen auf kompakte Felder übernehmen.

Wie bereits in der Einleitung bemerkt, muss man sich mit den Transformationsmatrizen und den Punktoperationen erst sehr spät und jenseits des Rahmens dieses Artikels tatsächlich beschäftigen. Für die Praxis genügt daher eine Punktdefinition mit drei (kompakten) Koordinaten vollauf. Um späteren Erweiterungen Raum zu lassen, definieren wir Punkte trotzdem vierdimensional, werden die vierte Dimension hier nicht explizit nutzen.

---

[7] Ausführlicher kann ich das hier leider nicht erläutern, doch sollte dies wohl genügen, zusammen mit anderen Informationen das Geschehen zu verstehen.

```
struct Point{
    Point():x(0),y(0),z(0),w(1){}
    Point(GLfloat xx, GLfloat yy, GLfloat zz):
                              x(xx),y(yy),z(zz),w(1){}
    GLfloat x,y,z,x;
};//end struct
```

Ein Grafikobjekt besteht im einfachsten Fall aus einer Anzahl von Punkten nebst ihren Darstellungseigenschaften, was zu der Containerdefinition

```
class PointBuffer {
public:
    ...
    virtual void draw() const;
    virtual void finish();
    vector<Point> points;
    Color color;
    int mode;
    bool finish_done;
};//end class
```

führt. Die Punkte und Eigenschaften werden auf öffentlichen Attributen gespeichert. Zusätzlich enthält die Klasse noch einige Methoden, die die Minima und Maxima der Koordinatenwerte ermitteln. Damit können die sichtbaren Koordinaten im GrafikSystem-Objekt abgeglichen werden, wobei allerdings zu beachten ist, dass es sich hierbei um die Koordinaten **vor** der Transformation handelt, die tatsächlichen Endkoordinaten also ganz andere Bereiche umfassen können.

Die virtuelle Methode `draw()` zeichnet innerhalb eines Display-Aufrufs die Punkte mit der angegebenen Farbe im spezifizieren Zeichenmodus und muss daher vom eigentlichen Grafikobjekt, dessen Bestandteil der Punktcontainer ist, aufgerufen werden:

```
void PointBuffer::draw() const{
    vector<Point>::const_iterator it;
    glBegin(mode);
    glColor4fv(&color.r);
    for(it=points.begin();it!=points.end();++it)
        glVertex4fv(&it->x);
    glEnd();
}//end function
```

`glBegin() ... glEnd()` definiert einen zusammenhängenden Punkteblock innerhalb des OpenGL-Systems, der in einer bestimmte Art (`mode`) zu zeichnen ist. Zeichenarten sind beispielsweise punktweises Zeichnen (GL_POINT), Verbinden der Punkte durch Linien (GL_LINE und weitere) oder Darstellung als Flächen (GL_FILL und andere), wobei der Nutzer des Puffers natürlich selbst dafür sorgen muss, dass die Punkte auch in der für die Darstellungsart notwendigen

Reihenfolge im Datenpuffer liegen. Sollen noch weitere Eigenschaften zur Darstellung kommen, die den Aufruf weitere Methoden im Zeichenblock erfordern, muss das in der `display()`-Methode des Objekts direkt realisiert werden, anstatt die `draw()`-Methode aufzurufen.[8]

Die Methode `finish()` dient der Vorbereitung des Datenobjekts im Rahmen der `init()`-Methoden der Objekte, das Attribut `finish_done` steuert den Aufruf der `finish()`-Methode. Wir werden erst in den Kapitel „Beleuchtungseffekte“ und „Flächendarstellungen“ genauer darauf eingehen, weil die Methoden erst dort benötigt werden.

13.3.2 Punkte auf einem Gitter und Flächendarstellung

Beliebige Flächen beschreibende Punkte werden meist in Gitterform abgelegt, d.h. die Fläche wird in $n_1 * n_2$ Felder eingeteilt und die Punkte zeilenweise in den Puffer eingetragen. Der Punkt P_{ik} wird dann an der Stelle $i * n_2 + k$ gespeichert, wodurch die Position eines Punktes im linearen Speicher eindeutig festliegt. Es ist sinnvoll, dies durch die Definition der speziellen Klasse

```
class GridBuffer: public PointBuffer {
public:
    GridBuffer();
    virtual bool set_dimension(int,int);
    inline Point const& get_grid_point(int n1,
                                       int n2)const
      {return points[n1*dim2+n2];}
    inline bool set_grid_point(int n1, int n2,
                               Point const& p)
      {points[n1*dim2+n2]=p; return true;}
    virtual void draw() const;
    int dim1,dim2;
};//endclass
```

zu unterstützen. Die Klasse erhält zwei zusätzliche Attribute `dim1, dim2`, die die Dimensionen der Punktmatrix enthalten und die nicht direkt, sondern durch die Methode `set_dimension(..)` vor Eintrag des ersten Punktes zu initialisieren sind. Die Methode ist so konstruiert, dass auch nachträglich die Dimensionen verändert werden können, ohne das alle Punkte neu zu berechnen sind (*die in die neue Größe passenden Punkte werden hinüber kopiert*). Die Erfassung und Ausgabe von Punkten erledigen `inline get_` - und `set_` - Methoden.

Aufgabe. Implementieren Sie die Methode entsprechend der Vorgaben.

[8] Einige Spezialisierungen des allgemeinen Typ stellen wir im Weiteren vor; der Fantasie des Nutzers ist natürlich bezüglich anderer Erweiterungen keine Grenze gesetzt.

Die `draw()`-Methode gestaltet sich hier je nach verwendetem Zeichenmodus aufwändiger. Wir beschränken uns hier auf zwei Modi, jedoch kann der Leser durchaus auch weitere Modi hinzufügen. Eine sehr häufig verwendete Methode zum Zeichnen von Freiformflächen ist die Unterteilung in Dreiecke, weil drei Punkte eine Ebene definieren und so weitere Eigenschaften im Zusammenhang mit Beleuchtungseffekten leicht zu berechnen sind. Wir sehen daher eine Ausgabe der Fläche als Drahtmodell aus Dreiecken oder als Dreiecksfläche vor.

```
void GridBuffer::draw() const{
   switch(mode){
     case GL_TRIANGLE_STRIP:
        glColor4fv(&color.r);
        for(int i=0;i<dim1-1;i++){
           glBegin(GL_TRIANGLE_STRIP);
           for(int j=0;j<dim2;j++){
              glVertex4fv(&get_grid_point(i,j).x);
              glVertex4fv(&get_grid_point(i+1,j).x);
           }//endfor
           glEnd();
        }//endfor
        break;

     case GL_LINE_STRIP:
        glColor4fv(&color.r);
        for(int i=0;i<dim1-1;i++){
           glBegin(GL_LINE_STRIP);
              glVertex4fv(&get_grid_point(i+1,0).x);
              glVertex4fv(&get_grid_point(i,0).x);
              for(int j=1;j<dim2;j++){
                 glVertex4fv(&get_grid_point(i,j).x);
                 glVertex4fv(&get_grid_point(i+1,j-1).x);
                 glVertex4fv(&get_grid_point(i+1,j).x);
                 glVertex4fv(&get_grid_point(i,j).x);
              }//endfor
           glEnd();
        }//endfor
        break;

     default:
        PointBuffer::draw();
   }//end switch
}//end function
```

Drahtmodell. Die Ausgabe als Drahtmodell erfolgt zweckmäßigerweise im Zeichenmodus GL_LINE_STRIP, bei dem die aufeinander folgenden Punkte jeweils durch Linien miteinander verbunden werden. Für den folgenden Algorithmus stelle man sich das Gitter in folgender Weise indiziert vor:

```
(2,0) -- (2,1) -- (2,2) -- (2,3) ...
  |         |         |         |
(1,0) -- (1,1) -- (1,2) -- (1,3) ...
  |         |         |         |
(0,0) -- (0,1) -- (0,2) -- (0,3) ...
```

Die Dreiecke lassen zeilenweise zeichnen, indem man mit den Linien

```
(i+1,0) - (i,0) - (i,1)
```

beginnt und bis zum Ende der Zeile (Zählvariable `j`) die Punktfolge

```
(i,j+1) - (i+1,j) - (i+1,j+1) - (i,j+1)
```

unter Inkrementieren der Zählvariablen `j` wiederholt und anschließend `i` um eins erhöht. Jede Zeile stellt dabei einen `glBegin - glEnd`-Block dar. Dabei wird jede Zeilenlinie doppelt gezeichnet.

Flachenmodell. Die Ausgabe als Fläche erfolgt im Modus GL_TRIANGLE_STRIP. In diesem Modus werden vom OpenGL-System jeweils drei aufeinander folgende Punkte als Dreieck interpretiert und gezeichnet. Anschließend wird der erste Punkt gestrichen und ein weiterer hinzugefügt, d.h. gezeichnet werden die Dreiecke:

```
(0,1,2),3,4,5,... → 0,(1,2,3),4,5,...
 → 0,1,(2,3,4),5... → ...
```

Zu beachten ist, dass das OpenGL-System die Reihenfolge der Punkte für die Ermittlung von Vorder- und Rückseite interpretiert. Entsteht aufgrund der Punktreihenfolge eine Drehung entgegen dem Uhrzeigersinn, handelt es sich im die Vorderseite. Bei der Reihenfolge der Punkte ist daher die Drehrichtung beizubehalten, will man später keine unangenehmen Überraschungen erleben.[9]

Auch hier erfolgt die Übergabe in Form zeilenweiser Blöcke mit der Punktreihenfolge

```
(i,j) - (i+1,j)
```

unter Inkrementierender Zählvariablen `j` und anschließendem Inkrementieren der anderen Zählvariablen.

13.3.3 Indizierte Punktlisten

Ein in der Praxis häufiger auftretender Fall besteht darin, dass ein Objekt, beispielsweise ein Haus, durch eine relativ geringe Anzahl von Punkten definiert werden kann, die Punkte nun aber mehrfach in verschiedenen Kombinationen und

[9] Das Interpretationsverhalten lässt sich auch individuell umschalten, was aber noch unangenehmer ist.

Zeichenmodi auftreten, um das Objekt grafisch darstellen zu können. Statt nun jede Fläche einzeln durch Punkte darzustellen (und dabei Eingabefehler zu riskieren), bietet es sich an, alle Objekteigenschaften zunächst in Liste zu erfassen und die einzelnen Objektteile in Form von Indexlisten zu beschreiben. Eine Indexklasse hat somit ein recht komplexes Innenleben:

```
class IndexBuffer: public PointBuffer {
public:
    void draw() const;
    void push_group(int size, int md, int c1, int i1, ...);
    vector<vector<int> > group;
    vector<int> modus;
    vector<Color> farbe;
};//end class
```

Neben der Punktliste, die nun jeden Punkt unabhängig von der Anzahl seiner Verwendungen nur einmal enthält, sind ebenfalls Listen für die Zeichenmodi und die Farben der verschiedenen Objektteile vorhanden. Die einzelnen Objektbestandteile werden mittels der Methode `push_group(..)` in der Indexliste abgelegt.

```
void IndexBuffer::push_group(int size, int md,
                             int c1, int i1, ...){
    vector<int> vi; int* pi = &md;
    for(int i=0;i<size+2;i++,pi++)
        vi.push_back(*pi);
    group.push_back(vi);
}//end function
```

Die Übergabeparameter sind die Anzahl der Punkte in der Gruppe, der Index des zu verwendenden Zeichenmodus, der Farbindex sowie die Liste der Punktindizes. Das Zeichnen erfolgt in der Methode `draw()`, wobei für jedes Element ein OpenGL-Block definiert wird

```
void IndexBuffer::draw() const{
    vector<vector<int> >::const_iterator git;
    vector<int>::const_iterator it;
    for(git=group.begin();git!=group.end();git++){
        it=git->begin();
        glBegin(modus[*it++]);
            glColor4fv(&farbe[*it++].r);
            for(;it!=git->end();it++){
                glVertex4fv(&points[*it].x);
            }//endfor
        glEnd();
    }//endfor
}//end function
```

13.4 Objekte und Szenen

In Grafikobjekten wird die Verbindung zwischen Datencontainern und Systemumgebung hergestellt. Die Grafikobjektklassen sind Basisklassen für Anwenderobjekte und stellen das Grunddatengerüst und die Grundfunktionalität zur Verfügung, die in den Anwenderklassen mit Daten zu füllen sind.

Szenen wiederum sind Sammlungen von mehreren Grafikobjekten innerhalb einer gemeinsamen Welt, die Informationen für die relative Positionierung der Objekte untereinander enthalten. Dabei kann auch der Fall auftreten, dass die Gesamtszene aus mehreren Einzelszenen besteht. Beispielsweise kann ein Auto eine Szene aus mehreren Objekten sein, die sich wiederum in einer größeren Szene, der Straße, bewegt.

13.4.1 Basisklasse

Die Basisklasse stellt die gemeinsame Wurzel von Grafikobjekten und Szenen dar.

```
struct ObjectBase {
    virtual void init(){};
    virtual void display(){};
    virtual void idle(){};

    Point            translation;   // Translationsvektor
    Point            scaling;       // Skalierungsvektor
    vector<Point>    rotation ;     // Rotationsachse
    vector<GLfloat>  phi;           // Drehwinkel
};//end struct
```

Sie enthält zum Einen einen Satz virtueller Methoden, die namensmäßig bereits in der Systembasisklasse auftreten und die gleiche Funktion besitzen. Sodann sind Attribute vorhanden, um das Objekt innerhalb seiner Teilwelt korrekt zu Positionieren. Zum Verständnis stellen Sie sich einen Würfel vor, dessen Mittelpunkt Nullpunkt aller Punktkoordinaten ist und der nun den verschiedenen Operationen unterworfen wird:

(a) Ein Skalierungsattribut sorgt für eine Streckung oder Stauchung der Objektkoordinaten, so dass aus dem Würfel ein Quader, beispielsweise ein Buch wird.
(b) Rotationen um verschiedene Achsen durch den Würfelnullpunkt bringen den Quader in eine bestimmte Ansicht. Da es etwas problematisch sein kann, für das Drehen in eine bestimmte Ansicht die korrekte Achse und den passenden Winkel anzugeben, kann die Drehung in mehrere Teildrehungen zerlegt werden.
(c) Abschließend wird der Würfelnullpunkt um eine bestimmte Strecke verschoben, so dass der Quader nun in Bezug auf den Nullpunkt eines übergeordneten Koordinatensystems positioniert ist.

Die gleichen Operationen finden auch statt, wenn es sich um eine komplette Szene handelt. Man könnte also vermuten, dass die `display()`-Methode gleich an dieser Stelle implementiert werden könnte, um später in Objekten und Szenen nur noch aufgerufen zu werden. Leider ist das nicht ganz so einfach, da die Transformationen zusammen mit den Punkten in OpenGL-Blöcken gekapselt werden müssen. Wir müssen daher zunächst eine Verbindung mit den Datentypen herstellen.

13.4.2 Objektklasse

Die Einrichtung von Objektklassen mit beliebigen Datenklassen ist mit Hilfe von Templates möglich, wobei wir nun Mehrfachvererbung nutzen, uns also an diesem Punkt von der Verwendung anderer Programmierumgebungen wie beispielsweise Java entfernen.

```
template <class DataClass>
class Object: public DataClass, public ObjectBase {

public:
  Object(){}
  ~Object(){}

  void init(){
    if(!DataClass::finish_done)
      DataClass::finish();
  }//end function

  void display(){
    glPushMatrix();
      glTranslatef(translation.x,translation.y,
                   translation.z);
      for(int i=0;i<rotation.size();i++)
        glRotatef(rotation[i].w,rotation[i].x,
                  rotation[i].y,rotation[i].z);
      glScalef(scaling.x,scaling.y,scaling.z);
      DataClass::draw();
    glPopMatrix();
    glFlush();
  }//end function;
};
```

Zu Implementieren sind an dieser Stelle die `init()`- und die `display()`-Methode, die, wie für Template-Klassen notwendig, direkt in der Klassendefinition untergebracht wird.

Die `init()`-Methode ruft in Abhängigkeit des Attributs `finish_done` die `finish()`-Methode des Datenteils auf. Beispiele für den Einsatz der `finish()`-Methode werden jedoch erst bei den Beleuchtungseffekten vorgestellt. Das Attribut

hilft, längere unnötige Initialisierungen zu vermeiden, wenn Objekte mehrfach definiert werden oder ihren Zustand als aktives Objekt mehrfach wechseln. Die Steuerung des Attributs obliegt der `finish()`-Methode selbst. Die `init()`-Methode wird voraussichtlich häufiger überschrieben; auf die Übernahme der Funktionalität ist dabei zu achten.

Um die Gesamtpositionierung eine Objektes zu erreichen, werden innerhalb der `display()`-Methode zunächst die Transformationen des Objektes in der Reihenfolge

1. Skalierung
2. sämtliche Drehungen in der Reihenfolge ihrer Eingabe
3. Translation

angewandt, dann die Transformationen der Teilszene usw. bis hin zur Transformation der Gesamtszene. Jede Transformation wird durch eine Transformationsmatrix bewerkstelligt.

Das Szenenobjekt ist das aktive Objekt, d.h. die Berechnung der Transformationsmatrix beginnt hier. Die Transformationsmatrix wird zunächst im System als Einheitsmatrix initialisiert (s.o.). Jede Transformationsanweisung wird von Rechts in diese Matrix hineinmultipliziert, so dass schrittweise die komplette Transformationsmatrix für eine Punktgruppe entsteht. Etwas anders ausgedrückt: die zuletzt definierte Transformationsanweisung im Programm wird zuerst ausgeführt. Um mit den oben definierten Transformationen die beabsichtigte Wirkung zu erreichen, ist daher zuerst die Translation zu definieren, gefolgt von den Rotationen und abgeschlossen von der Skalierung, d.h. zuerst wird das Objekt skaliert, dann gedreht und abschließend an seinen Bestimmungsort geschoben.

Nun sind die Transformationen für ein Objekt ausschließlich auf dieses Objekt anzuwenden, d.h. die für dieses Objekt berechnete Transformationsmatrix wäre beim nächsten Objekt falsch. Um jedes Objekt korrekt zu zeichnen, wird die aktuelle Transformationsmatrix daher zunächst durch `glPushMatrix()` auf einem Stack gesichert, anschließend die Transformationen für die Objektpunktgruppe vorgegeben und die Punkte übertragen. Bei Übertragung der Transformationen wird die aktuelle Matrix neu berechnet, um die Punkttransformationen durchführen zu können. Abschließend mit `glPopMatrix()` die Transformationsmatrix vor Zeichnen des Objekts wiederhergestellt, um auch das nächste Objekt korrekt in der Hauptwelt transformieren zu können.

13.4.3 Szenen

In einer Szene wird schließlich alles vereinigt, und eine Szene übernimmt schließlich die Rolle des aktiven Objekts. Die Szenenklasse erbt somit von des Systemklasse, enthält eine Liste der Objekte und muss schließlich, da auch Szenen in Szenen auftreten können, selbst Eigenschaften eines Objektes aufweisen. Da Objekte unterschiedlichen Klassen angehören (können) und die Eigenschaften der virtuellen Vererbung benötigt werden, müssen wir beim Entwurf von Szenen mit Zeigern

arbeiten. Zur Vermeidung von Problemen mit nicht freigegebenem Speicherplatz von Zeigerobjekten machen wir zusätzliche eine Anleihe bei SmartPointern mit integrierter Mehrfachreferenz. Eine Szene besitzt aufgrund der Vorbemerkungen das recht komplexe Vererbungsmuster:

```
class Scene: public GrafikSystem,
             public list<Ptr_MR<ObjectBase> >,
             public ObjectBase {
public:
    Scene();
    void init();
    void display();
    void idle();
    void mouse_motion(int x, int y);
    void mouse(int,int,int,int);
    typedef Ptr_MR<ObjectBase> PObject;
};//end class
```

Innerhalb der Szenenklasse implementieren wir die drei Methoden `init()`, `display()` und `idle()` (ggf. auch die Timerfunktion), die jeweils die entsprechenden Methoden der einzelnen Objekte aufrufen müssen.

```
void Scene::display(){
    list<Ptr_MR<ObjectBase> >::iterator oit;
    glPushMatrix();
        glTranslatef(translation.x,translation.y,
                     translation.z);
        for(int i=0;i<rotation.size();i++){
            glRotatef(rotation[i].w,rotation[i].x,
                      rotation[i].y,rotation[i].z);
        }//endfor
        glScalef(scaling.x,scaling.y,scaling.z);
        for(oit=begin();oit!=end();++oit){
            (*oit)->display();
        }//endfor
    glPopMatrix();
}//end function

void Scene::idle(){
    list<Ptr_MR<ObjectBase> >::iterator oit;
    for(oit=begin();oit!=end();++oit){
        (*oit)->idle();
    }//endfor
}//end function

void Scene::init(){
    list<Ptr_MR<ObjectBase> >::iterator oit;
```

```
    for(oit=begin();oit!=end();++oit){
        (*oit)->init();
    }//endfor
}//end function
```

`idle()` ruft nacheinander alle `idle()`-Methoden in der Liste auf, `init()` macht das Gleiche für die `init()`-Methoden, und `display()` wiederholt das bei den Objekten beschriebene Aufrufschema mit dem Unterschied, dass nun `display()` statt `draw()` aufgerufen wird.

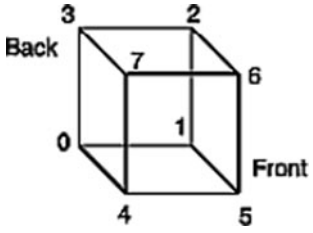
Im Gegensatz zu den Objektklassen, die häufiger überschrieben werden müssen, muss die Szenenklasse wohl in den seltensten Fällen tatsächlich spezialisiert werden. Einzelne Erweiterungen sind zentral durchführbar, so dass trotz des komplexen Vererbungsschemas kaum mit Problemen zu rechnen ist.

13.4.4 Objektbibliotheken

Konkrete Objekte erben von der Objektklasse. Einen Eindruck der Vorgehensweise vermittelt die Klasse

```
class Cubus: public Object<IndexBuffer>
```

mit folgendem Punkteschema des Würfels. Die Koordinaten werden jeweils auf ± 1 gesetzt, so dass das Zentrum des Würfels den Koordinatenursprung bildet.



Im Konstruktor wird die Punktliste, die Farben für jede Fläche und die Punktgruppen für jedes Quadrat erfasst.

```
Cubus::Cubus(){
    points.push_back(Point(-1,-1,-1));
    points.push_back(Point(1,-1,-1));
    ...
    farbe.push_back(Color(0.5,0,0));
    farbe.push_back(Color(0,0.5,0));
    ...
    modus.push_back(GL_QUADS);

    push_group(4,0,4,1,2,6,5);  // rechts
    push_group(4,0,3,0,1,5,4);  // unten
    ...
```

```
    push_material_group(4,GL_FRONT_AND_BACK,
                           GL_AMBIENT_AND_DIFFUSE,4,
                           GL_FRONT_AND_BACK,
                           GL_SHININESS,7,
                           GL_FRONT_AND_BACK,
                           GL_SPECULAR,8,
                           GL_FRONT_AND_BACK,
                           GL_EMISSION,6);
    ...
    simple=true;
}//end function
```

Das Beispiel umfasst jeweils nur einige der acht Punkte bzw. sechs Gruppen, zusätzlich sind weitere Eigenschaften erfasst, zu denen wir weiter unten noch kommen werden.

In der `idle()`-Funktion können die Drehwinkel erhöht werden. Die Klasse enthält bereits einige Parameter, die für die Beleuchtungssteuerung benötigt werden. Abbildung 13.3 enthält vier Würfelobjekte in zwei Szenen zu je zwei Würfeln, die in einer Zentralszene dargestellt werden.

Aufgabe. Versuchen Sie, das Übungsbeispiel vollständig zu implementieren.

WICHTIG! Beim Entwurf komplexer Szenen werden voraussichtlich eine Vielzahl von Objekt-, Szenen- und Beleuchtungsklassen definiert und montiert. Hierbei ist unbedingt zu beachten:

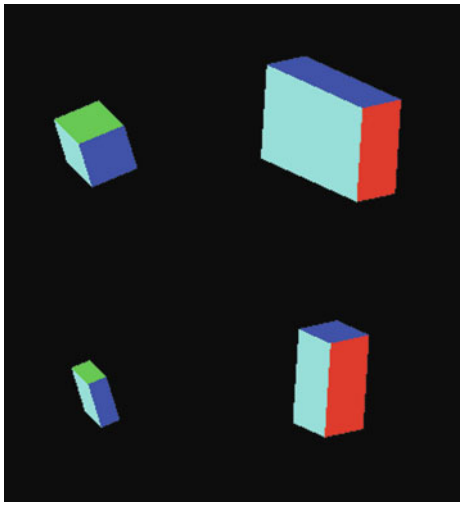


Abb. 13.3 Übungsbeispiel

(a) In den Basisklassen der Datenobjekte werden im Rahmen der Initialisierung die `finish()`-Methoden verwendet, um fehlende Daten wie Normalvektoren zu berechnen. Werden die Methoden nicht ausgeführt, kommt es zu Laufzeitfehlern. Zu ähnlichen Effekten kann es fallweise auch in anderen Objekten kommen. Die Initialisierungskette darf deshalb nicht unterbrochen werden.

Leider stellt C++ keinen Mechanismus zur Verfügung, automatisch Methoden der Elternklassen auszuführen, und die Berücksichtigung eines entsprechenden Template-Wrap-Arounds schien uns hier zu aufwändig (vergleiche aber auch ein späteres Kapitel). In überschriebenen `init()`-Methoden muss daher selbst darauf geachtet werden, die `init()`-Methode der Elternklasse (rekursiv) an einem geeigneten Punkt aufzurufen:

```
struct Scene1: public Scene {
    void init(){
        ....;
        Scene::init();
    }//end function
```

(b) Die `idle()`-Methoden dienen häufig zur Dynamisierung von Szenen. Auch hier ist ein rekursiver Aufruf sämtlicher Methoden in den Basisklassen vorgesehen. Durch eigene Klassen definierte `idle()`-Methoden sollten daher wie unter (a) beschrieben die `idle()`-Methoden der Basisklassen aufrufen.

(c) Sofern weitere Ereignisfunktionen szenenweit verwendet werden sollen, ist das Klassenmodell entsprechend anzupassen.

13.5 Beleuchtungseffekte

13.5.1 Grundlagen

Wie ein Objekt bei Beleuchtung wirkt, hängt von den Lichtquellen, den Oberflächenbestandteilen des Objektes und seiner Lage relativ zu anderen Objekten ab. Beginnen wir bei den Lichtquellen:

- **Diffuses Licht** beschreibt eine allgemeine Helligkeit ohne besondere Richtungseigenschaften und entsteht beispielsweise durch Rückstreuung von Licht an Wolken oder anderen Objekten.
- **Direktionales Licht** kommt aus einer bestimmten Richtung, die aber unabhängig von der Lage der Objekte im Raum für alle Objekte die gleiche ist. Dies entspricht der Wirkung von Sonnenlicht.
- **Spotlicht** hat eine genau definierte Lichtquelle im Raum mit einem definierten Öffnungswinkel und wirkt sich auf jedes Objekt anders aus.

Die Objekte reagieren auf Beleuchtung aufgrund ihre Oberflächeneigenschaften:

- Die **Reflexion** wirft das auftreffende Licht wieder in den Raum zurück und wird in zwei Komponenten untergliedert:
  - Die **spiegelnde Reflexion** wirft den auftreffenden Lichtstrahl symmetrisch zum Flächennormalenvektor zurück. Die Beleuchtungsintensität am Standort der Kamera nimmt schnell ab, wenn der Winkel des Kamerarichtungsvektors mit Reflexionsvektor sich vergrößert.
  - Die **diffuse Reflexion** wirft ein von Einfallwinkel nur schwach abhängiges Signal gleichmäßig in alle Richtungen.

Die realen Eigenschaften sind eine Mischung aus beidem.

- Die **Absorption** beschreibt die Verringerung/Veränderung des Lichtes aufgrund der Reflexion und teilt sich in einen absoluten und einen farbabhängigen Anteil auf.
- Die **Transmission** beschreibt den durch die Oberfläche durchtretenden Lichtanteil, wobei auch hier wieder Farbeffekte zu beachten sind.

Diese Grundeigenschaften führen zu weiteren Effekten, beispielsweise

- werden Objekte nicht nur von der Lichtquelle beleuchtet, sondern auch von den von anderen Objekten reflektierten Lichtstrahlen,
- spiegeln sich Objekte bei hoher Reflexion in den Oberflächen anderer Objekte, oft sogar in Form von Mehrfachspiegelungen,
- brechen transparente Objekte das Licht, so dass dahinter liegende Objekte verzerrt werden,

und anderes mehr. Die Liste deutet schon an, dass vom OpenGL-System bei weitem nicht alles übernommen werden kann, sondern lediglich einige Basisbeleuchtungsschemata bearbeitet werden können. Um eine fotorealistisch ausgeleuchtete Szene zu erhalten, muss ein so genanntes Ray Tracing durchgeführt werden, bei dem entweder von der Projektionsebene oder von der/den Lichtquellen ausgehend jeder Strahl mit entsprechenden Aufspaltungen bis zum jeweiligen Gegenpart durchgerechnet wird. Ray Tracing berücksichtigt hauptsächlich die Anteile der Totalreflexion, während bei diffusen Lichtanteilen Näherungsmethoden angewandt werden. Wir werden uns hier auf die von OpenGL angebotenen Lichtmodi beschränken.

13.5.2 Lichtquellen

OpenGL erlaubt die Definition von mindestens acht unabhängigen Lichtquellen. Lichtquellen in unserem Sinn sind spezielle Objekte, erben also von `ObjectBase`:

```
class LightSource: public ObjectBase {
public:
    enum SourceType {spot, sun} sourceType;

    LightSource(SourceType st);
```

```
    ~LightSource();
    void init();
    void display();

    int light_nr;
    Color ambient;
    Color specular;
    Color diffuse;
    Point position;

    Point spot_direction;
    GLfloat spot_angle;
};//end class
```

Die Klassendefinition umfasst einige neue Attribute und überschreibt die Methoden `init()` und `display()`. Um den Programmierer von der Beobachtung der Numerierung der installierten Lichtquellen zu entbinden, fügen wir im Konstruktor eine automatische Zuweisung ein, die jedoch aufgrund der öffentlichen Attribute auch jederzeit von Hand überschrieben werden kann.

```
static bool light_active[8] =
   { false,false,false,false,false,false,false,false};
static int  light_number[8] =
   {GL_LIGHT0,GL_LIGHT1,GL_LIGHT2,GL_LIGHT3,GL_LIGHT4,
    GL_LIGHT5,GL_LIGHT6,GL_LIGHT7};

LightSource::LightSource(SourceType st):sourceType(st){
    int i;
    for(i=0;i<8;i++)
        if(!light_active[i]) break;
    light_nr=light_number[i];
    light_active[i]=true;
    ambient=Color(0,0,0,1);
    diffuse=Color(0.3,0.3,0.3,1);
    specular=Color(1,1,1,1);
    switch(st){
        case sun:
            position=Point(1,0,1,0);
            break;
        case spot:
            position=Point(0,0,0,1);
            spot_direction=Point(1,0,0,1);
            spot_angle=180;
            break;
    }//endswitch
}//end
```

Jede Lichtquelle kann drei Lichtanteile zur Beleuchtung beisteuern (siehe oben), die in drei Farbvektoren definiert sind. Den Vektoren werden im Konstruktor gebräuchliche Werte für weißes Licht zugewiesen. Bei farbiger Beleuchtung sind die Vektoren entsprechend anzupassen.

Die Positionsangabe der Lichtquellen weicht etwas vom bisherigen Schema ab, da direktionales Licht formal in unendlicher Entfernung positioniert wird. Dies kann in homogenen Koordinaten durch $w = 0$ erreicht werden. Für die Koordinatenangaben von Lichtquellen benötigen wir daher erstmalig vierdimensionale Raumkoordinaten.

Während Sonnenkoordinaten im Prinzip die 3D-Strahlrichtung der Lichtquelle im Unendlichen enthalten, muss bei Spotleuchten, die normale Raumkoordinaten mit $w = 1$ besitzen, zusätzlich noch die Strahlungsrichtung und der Öffnungswinkel angegeben werden.[10] Neben den Attributen definieren wir noch zwei Lichtmodi für Sonnen- und Spotlicht und weisen den Attributen im Konstruktor plausible Standardwerte zu.

In der `init()`-Methode werden die Farbparameter gesetzt und die Lichtquelle aktiviert.

```
void LightSource::init(){
    glDisable(GL_LIGHTING);
    glShadeModel (GL_SMOOTH);

    glLightfv(light_nr, GL_AMBIENT, &ambient.r);
    glLightfv(light_nr, GL_DIFFUSE, &diffuse.r);
    glLightfv(light_nr, GL_SPECULAR, &specular.r);
    glLightfv(light_nr, GL_POSITION, &position.x);
    if(sourceType==spot){
        glLightfv(light_nr,GL_SPOT_DIRECTION,
                  &spot_direction.x);
        glLightf(light_nr,GL_SPOT_CUTOFF,spot_angle);
    }//endif

    glEnable(GL_LIGHTING);
    glEnable(light_nr);
};
```

Auch die `display()`-Methode unterscheidet sich nur unwesentlich von den bisherigen `display()`-Methoden, nur dass anstelle der Punktkoordinaten die vierdimensionalen Lichtquellenkoordinaten gesetzt werden. Die Skalierung kann entfallen.

```
void LightSource::display(){
    glPushMatrix();
```

[10] Zusätzlich wäre auch noch ein Abschwächungsfaktor notwendig, der die Abnahme der Lichtintensität in Abhängigkeit von der Entfernung zum Objekt angibt. Wir behandeln diesen Effekt aber an dieser Stelle nicht.

```
        glTranslatef(translation.x,translation.y,
                     translation.z);
        for(int i=0;i<rotation.size();i++){
            glRotatef(rotation[i].w,rotation[i].x,
                      rotation[i].y,rotation[i].z);
        }//endfor
        glLightfv(light_nr,GL_POSITION,&position.x);
        if(sourceType==spot){
            glLightfv(light_nr,GL_SPOT_DIRECTION,
                      &spot_direction.x);
            glLightf(light_nr,GL_SPOT_CUTOFF,spot_angle);
        }//endif
    glPopMatrix();
}//end function
```

OpenGL-intern werden Lichtquellen genauso behandelt wie andere Objekte, d.h. sie unterliegen den Transformationsmatrizen. Um Lichtquellen zu Positionieren können daher die speziellen Koordinaten der Lichtquellen (Positionsvektor, Richtungsvektor des Spots) oder die allgemeinen Transformationsparameter des Basisobjekts verwendet werden. In Bezug auf die Gesamtszene ist ebenfalls darauf zu achten, wo eine Lichtquelle positioniert wird, um den gewünschten Effekt – statische Lichtquelle, objektgebundene Lichtquelle, frei bewegliche Lichtquelle – zu erhalten.

13.5.3 Objekteigenschaften

Betrachtet man eine mit einer Lichtquelle ausgeleuchtete Szene, wird man noch nicht das folgende Bild sehen:

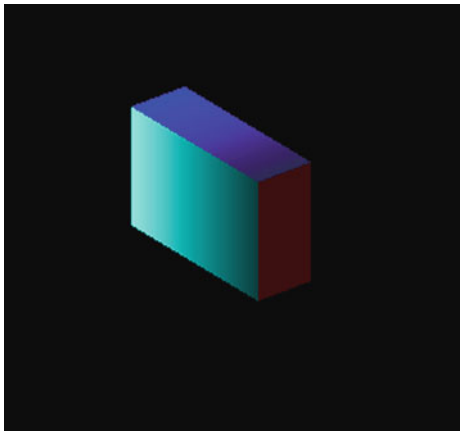


Abb. 13.4 Quader mit Spotbeleuchtung

Zum Einen wird die Farbe fehlen, zum Anderen wird man feststellen, dass die Flächen unabhängig von ihrer Orientierung zur Lichtquelle stets die gleiche Reflexion aufweisen und nicht bei flacheren Winkeln dunkler werden.

Für einfache Beleuchtungszwecke kann die in den Datensätzen der Objekte angegebene Farbe aktiviert werden, indem nach Setzen der Farbe die Befehle

```
glColorMaterial(GL_FRONT_AND_BACK,
                GL_AMBIENT_AND_DIFFUSE);
glColorMaterial(GL_FRONT_AND_BACK,GL_SPECULAR);
glEnable(GL_COLOR_MATERIAL);
```

aktiviert werden. Die Befehlssequenz darf nicht innerhalb der **glBegin()..glEnd()**-Klammer stehen. Wir berücksichtigen diese Beleuchtungsmethode in spezialisierten Datenklassen, wobei wir hier die Indexklasse als Beispiel verwenden:

```
class LightIndexBuffer: public IndexBuffer {
public:
    LightIndexBuffer();
    void draw() const;
    void finish();
    void push_material_group(int size,
                         int face, int mode, int
    value, ...);
    bool simple;
    vector<Point> normal;
    vector<vector<Triple> > material;
};//end class
```

Sie enthält zwei weitere Container für die Normalvektoren sowie für Gruppen von Materialeigenschaften. Über das Attribut `simple` kann zwischen der einfachen Farbsteuerung und der komplexen Farbsteuerung umgeschaltet werden. Wird die einfache Farbsteuerung verwendet, brauchen Materialgruppen nicht zugewiesen zu werden, d.h. die in Abschnitt 3.3 beschriebenen Parameter genügen.

Die Normalvektoren können manuell vorgegeben oder automatisch berechnet werden. Hier sind drei Hifsfunktionen definiert.

```
inline Point vec_diff(Point const& p1, Point const& p2){
    return Point(p1.x-p2.x,p1.y-p2.y,p1.z-p2.z);
}//end function

inline Point vec_prod(Point const& p1, Point const& p2){
    return Point(p1.y*p2.z-p1.z*p2.y,
                 p1.z*p2.x-p1.x*p2.z,
                 p1.x*p2.y-p1.y*p2.x);
}//end function

inline Point vec_norm(Point const& p1){
    GLfloat n=sqrt(p1.x*p1.x+p1.y*p1.y+p1.z*p1.z);
```

```
    return Point(p1.x/n,p1.y/n,p1.z/n);
}//end function
```

Für die automatische Berechnung definieren wir die Methode `finish()`, die auch in den Basisklassen nachdefiniert werden muss und ähnlich wie `draw()` in den `display()`-Methoden in den`init()`-Methoden aufgerufen wird. Mittels dreier Methoden für die Berechnung von Abstandsvektoren von Punkten, für die Berechnung des Kreuzprodukt zweier Vektoren und für die Normierung von Vektoren berechnet die `finish()`-Methode die Flächennormale aus den ersten drei Punkten der Punktgruppe. Hierbei wird vorausgesetzt, dass alle Punkte einer Gruppen in einer Ebene liegen.

```
void LightIndexBuffer::draw() const{
    vector<vector<int> >::const_iterator git;
    vector<int>::const_iterator it;
    vector<Point>::const_iterator pt;
    vector<Triple>::const_iterator tit;

    for(git=group.begin(),pt=normal.begin();
        git!=group.end();git++,pt++){
        it=git->begin();
        if(simple){
            glColor4fv(&farbe[*(it+1)].r);
            glColorMaterial(GL_FRONT_AND_BACK,
                            GL_AMBIENT_AND_DIFFUSE);
            glColorMaterial(GL_FRONT_AND_BACK,
                            GL_SPECULAR);
            glEnable(GL_COLOR_MATERIAL);
        }else{
            for(tit=material[*(it+1)].begin();
                tit!=material[*(it+1)].end();++tit){
                glMaterialfv(tit->first,
                     tit->second,&farbe[tit->third].r);
            }//endfor
        }//endif
        glBegin(modus[*it++]); it++;
            glNormal3fv(&pt->x);
            for(;it!=git->end();it++){
                glVertex4fv(&points[*it].x);
            }//endfor
        glEnd();
    }//endfor
}//end function

void LightIndexBuffer::finish(){
    vector<vector<int> >::const_iterator git;
    vector<int>::const_iterator it;
```

```
    vector<Point>::const_iterator pt;
    normal.clear();
    for(git=group.begin();git!=group.end();git++){
        normal.push_back(
            vec_norm(
                vec_prod(
                    vec_diff(points.at(git->at(3)),
                             points.at(git->at(2))),
                    vec_diff(points.at(git->at(4)),
                             points.at(git->at(3))))));
    }//endfor
    finish_done=true;
}//end function
```

WICHTIG! Zu jeder Indexgruppe wird bei dieser Vorgehensweise ein Normalvektor definiert. Die Containerindizes von Gruppe und Normalvektor müssen gleich sein.

Materialeigenschaften werden mit der OpenGL-Methode

```
glMaterialfv(GL_FRONT,GL_DIFFUSE,&color);
```

an das Grafiksystem übertragen, wobei zwischen Vorder- und Rückseite einer Fläche, diffusem, ambienten und reflektierten Licht, der Reflexionsstärke sowie einem Eigenleuchten unterschieden werden kann. Im Materialcontainer können daher mehrere Zahlentripel abgelegt werden, wobei die ersten beiden Parameter Seite und Modus, der dritte den Index in der Farbtabelle, die bei dieser Steuerungsart ja nicht mehr direkt verwendet wird, angibt:

```
vector<Triple>::const_iterator tit = material[0].begin();
glMaterialfv(tit->first,tit->second,
             &farbe[tit->third].r);
```

Die `draw()`-Methode der Klasse ist damit eine simple Erweiterung der Mutterklasse.

Die Erweiterung der Indexdatenklasse zur Berücksichtigung von Lichteffekten gehört sicherlich zu den komplexeren. Bei den anderen Datenklassen muss in der Regel weniger Aufwand getrieben werden; bei Verwendung von bereits in den OpenGL-Bibliotheken definierten Körpern (Würfel, Kugel und andere) entfällt in der Regel auch die Definition und Berechnung der Flächennormalen.

13.6 Perspektivische Projektion

13.6.1 Grundlagen der perspektivischen Darstellung

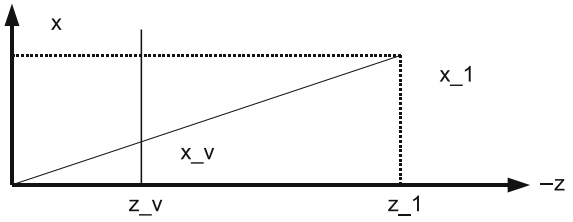
Alle bisherigen Matrizen waren Transformationsmatrizen, d.h. den Punkten im dreidimensionalen Raum wurden aufgrund der Transformation neue Koordinaten im dreidimensionalen Raum zugewiesen. Das Sichtfenster, hier der Bildschirm, ist aber

zweidimensional, d.h. alle Punkte müssen auf den Bildschirm projiziert werden, indem die Z-Koordinate gewissermaßen wegrationalisiert wird. Im einfachsten Fall, der orthogonalen Projektion werden von allen Punkten nur die XY-Koordinaten verwendet, wobei außerhalb des Fensters liegende Punkte einfach abgeschnitten werden. Die Z-Koordinate wird aber nicht einfach abgeschnitten, sondern zur Auswertung von Überdeckungen herangezogen.

Die orthogonale Projektion entspricht aber nicht der Realität, da weiter entfernte Objekte kleiner erscheinen als gleich große nahe. Die XY-Koordinaten ändern sich bei der Projektion daher aufgrund einer perspektivischen Projektion. In OpenGL ist die perspektivische Projektion nicht sonderlich aufregend, da sie durch einfache Funktionsaufrufe realisiert wird. Es lohnt sich aber doch, einmal auf die Theorie dahinter zu schauen. Hier kommt nämlich erstmals die bislang unterschlagene vierte projektive Koordinate zum Einsatz. Da die Darstellung im 3D erfolgt, müssen alle Punkte, die zur Darstellung gelangen, den gleichen Koordinatenwert der vierten Koordinate aufweisen. Vereinbarungsgemäß gilt die Normierungsvorschrift

$$\vec{v}_{3D} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, Normierung: \vec{w}_p = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \vec{w}_{3D} = \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$$

Für die Blickrichtung wird vereinbarungsgemäß immer die negative Z-Achse verwendet. Bei einer perpektivischen Projektion ändern sich die scheinbaren X- und Y-Koordinaten eines Punktes in Abhängigkeit von seiner Z-Koordinate:



Sei ein Punkt mit den Koordinaten (x_1,y_1,z_1) gegeben und z_v die Projektionsebene. Die scheinbaren neuen Koordinaten ergeben sich nach dem Strahlensatz zu

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \vec{v}\,' = \begin{pmatrix} x/y * z_v \\ y/z * z_v \\ z \end{pmatrix}$$

Um das nun für eine große Anzahl von Punkten effektiv umzusetzen, ist die Überführung in eine Projektionsmatrix sinnvoll. Eine Division ist allerdings innerhalb einer linearen Transformation nicht machbar, jedoch können wir die Normierungsvorschrift für die 3D-Darstellung von Punkten mit homogenen Koordinaten nutzen. Die Projektionsmatrix erhält dann folgende Form

$$P = \begin{pmatrix} z_v & 0 & 0 & 0 \\ 0 & z_v & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & b \end{pmatrix}, \vec{v} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \rightarrow \vec{v}\,' = \begin{pmatrix} x * z_v \\ y * z_v \\ a * z + b \\ z \end{pmatrix}$$

Das Ergebnis wirkt auf den ersten Blick vermutlich nicht sonderlich hilfreich, weshalb einige Erläuterungen angebracht seien. Die perspektivische Transformationsmatrix ist die letzte in der Reihe der Projektionsmatrizen und ist problemlos in die Gesamtprojektionsmatrix für die Punkte zu integrieren. Bevor das System nun tatsächlich mit der Darstellung anfängt, ist zu entscheiden, welche Punkte dem Clipping unterliegen, also nicht dargestellt werden, da sie außerhalb des sichtbaren Volumens liegen bzw. von anderen Bildkomponten überdeckt werden. Da bei einer Abschlussnormierung die z-Komponente der Transformation $(a * z + b) \rightarrow (a + b/z)$ unterliegt, kann bei geschickter Wahl von a und b ein großer Teil dieses Geschäftes erledigt werden, ohne die Normierung tatsächlich durchführen oder überhaupt viel rechnen zu müssen. Nur für die danach noch übrigen Punkte wird die Normierung durchgeführt, aber auch dann nur für die Berechnung der Koordinaten (x,y), da dies für die Ermittlung der restlichen Überdeckungen ausreicht.

Auch die perspektivische Projektion wird natürlich wieder dem OpenGL-System überlassen. Sie ersetzt die orthogonale Projektion und ist damit in unserem Klassenschema in der `GrafikSystem`-Klasse anzusiedeln. Es sei jedoch darauf hingewiesen, dass diese Projektionsweise nur eine Näherung darstellt, da die Z-Koordinate nicht alleine für die Entfernung eines Objektes vom Ursprung verantwortlich ist. An den Rändern der Projektionsebene kann es daher zu Verzerrungen kommen.

13.6.2 Projektionsdefinition

Das sichtbare Volumen wird in Form eines Pyramidenschnitts wird durch den Funktionsaufruf

```
glFrustum(xmin,xmax,ymin,ymax,zmin,zmax);
```

an das OpenGL-System übertragen. Die Sichtrichtung ist vereinbarungsgemäß die -Z-Richtung, XY-Koordinaten geben die Größe des vorderen Projektionsbildschirms an, die Z_{min}-Koordinate den Abstand der Projektionsebene vom Nullpunkt und die Z_{max}-Koordinate die Tiefe des dargestellten Raumes. Das OpenGL-System kann nun zunächst alle Objektteile außerhalb dieses Volumens ausschneiden, Überdeckungen entfernen, die auch ohne Projektion vorhanden wären, die oben beschriebene Projektion auf die noch vorhandenen Objekte anwenden und nach Beseitigung der restlichen Überdeckungen das Ergebnis darstellen.

Das Ergebnis wird bei den ersten Versuchen voraussichtlich ein weitgehend leeres Bild sein. Dies ist darauf zurückzuführen, dass bei einer orthogonalen Projektion die Objekte in der Nähe des Koordinatenursprungs angesiedelt und das Fenster im Positiven liegt, nun aber die Projektionsfläche selbst bereits negativen Koordinaten besitzt und ein Großteil der Objekte gewissermaßen hinter dem Sichtfenster liegen. Die Szene muss daher in zunächst in den Pyramidenstumpf verschoben werden.

Um das nun nicht zu kompliziert zu machen und auch gleich noch einige andere Parameter zu berücksichtigen, stellt die glu-Bibliothek zwei Funktionen zur Verfügung, die das etwas vereinfachen. Die Methode

```
gluLookAt(...)
```

definiert eine Kamera und besitzt 9 Übergabeparameter, die die Korrdinaten

- des Kamerastandortes (die Szene wird um das Negative dieses Vektors verschoben),
- der Blickrichtung (die Szene wird so um die X- und die Y-Achse gedreht, dass die Blickrichtung auf die negative Z-Achse fällt) und
- des Oben-Vektors angeben (die Szene wird um die Z-Achse gedreht, so dass der Oben-Vektor mit der Y-Achse koinzidiert).

Hierfür sehen wir im Basisobjekt einen Satz Parameter vor, der in der `display()`-Methode nach Aktivierung der MODELVIEW-Matrix aktiviert wird. Die MODELVIEW-Matrix wird mit diesen Daten geladen, so dass bei den Transformationen alle Objekte sofort auf die richtige Position geschoben werden. Die Änderung der Parameter im aktiven Objekt können so Kamerafahrten realisiert werden.

Als zweite Methode stellt die glu-Bibliothek

```
gluPerspective(fovy,aspect,zNear,zFar)
```

zur Verfügung (hier nicht verwendet). `fovy` gibt den Öffnungswinkel, bezogen auf die Y-Achse an, `aspect` das Verhältsnis Y/X und die restlichen Parameter die Tiefe des Pyramidenstumpfes an. Die resultierende Projektionsmatrix ist

$$\begin{pmatrix} \frac{fovy}{aspect} & 0 & 0 & 0 \\ 0 & fovy & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2*zFar*zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Die Verbindung mit der `glFrustum()`-Methode ist damit klar, die glu-Methode ist aber intuitiv einfacher zu bedienen, da sie ergänzend zur Kameraposition gewissermaßen das verwendete Objektiv beschreibt.

HINWEIS! Bei Umschalten zwischen dem orthogonalen und dem perspektivischen Modus sind einige Änderungen an der Projektionsmatrix vorzunehmen. Die Umschaltung wird daher zweckmäßigerweise mit `swap_perspective_view()` vorgenommen.

13.7 Flächendarstellungen

13.7.1 Texturen

Mit einfachen, aus ebenen Polygonflächen zusammengesetzten oder durch die glut-Bibliothek definierten Körpern ist es häufig nicht getan, wenn die Formen komplizierter werden, und auch eine einfache Mauer wird kompliziert, wenn der Aufbau aus Ziegelsteinen mit den bisher vorhandenen Mitteln realisiert werden soll. Statt aber nun alles als Vektorgrafik vorzugeben, kann man auch Teile der Objekte in Form einer Bitmap, d.h. gewissermaßen eines Fotos eines fertigen Objektes, vorgeben und diese darstellen lassen.

Für Texturen definieren wir eine weitere Datenklasse, die von einer der Punktcontainerklassen erbt:

```
class Texture: public PointBuffer {
public:
    Texture();
    ~Texture();
    void finish();
    void draw();
    int w,h;
    string filename;
    bool clamp;
protected:
    GLuint handle;
};//end class
```

Texturen werden in der Regel aus Dateien eingelesen. Die meisten Bildbearbeitungsprogramme bieten das Dateiformat .raw an, mit dem die reinen Bitmap-Informationen ohne weitere Kopfinformationen in einer Datei abgespeichert werden können. Die Attribute der Textur-Klasse übernehmen den Dateinamen sowie Höhe und Breite des Bildes, da dies nicht aus den Dateiinformationen hervorgeht.

Die `finish()`-Methode ist für das Einlesen der Textur zuständig.

```
void Texture::finish(){
    if(handle==-1){
        glGenTextures(1,&handle);
        cout << "Texture=" << handle << endl;
    }//endif
    ifstream ifs(filename.c_str(),ios::binary);
    if(ifs.is_open()){
        char* buffer = new char[w*h*3];
        ifs.read(buffer,w*h*3);
        ifs.close();
        glEnable(GL_TEXTURE_2D);
```

```
        glBindTexture(GL_TEXTURE_2D,handle);
        gluBuild2DMipmaps(GL_TEXTURE_2D,3,h,w,GL_RGB,
                          GL_UNSIGNED_BYTE,buffer);
        delete[] buffer;
    }//end
}//end function
```

Hierzu wird zunächst ein Handle für einen Texturspeicher im OpenGL-System angefordert sowie die Texturdarstellung für eine Flächentextur aktiviert. Nach Aktivieren der Textur erfolgt die Übergabe der Bitmap an das OpenGL-System, dass die Textur für die Ausgabe vorbereitet und intern speichert. Der Speicher im Arbeitsprogramm kann wieder freigegeben werden, bei Wegfall der Textur selbst auch deren Speicher im OpenGL-System.

In der `draw()`-Methode werden nach Aktivieren der Textur eine Reihe von Arbeitsparametern gesetzt, mit denen definiert wird, wie mit der Textur unter Berücksichtigung von Farben und Licht umzugehen ist.

```
void Texture::draw(){
    vector<Point>::const_iterator it;
    glColor4fv(&color.r);
    glBindTexture(GL_TEXTURE_2D,handle);
    glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_MIN_FILTER,
              GL_MODULATE);
    glTexParameterf(GL_TEXTURE_2D,
        GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAREST);
    glTexParameterf(GL_TEXTURE_2D,
        GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                    (clamp?GL_CLAMP:GL_REPEAT));
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    (clamp?GL_CLAMP:GL_REPEAT));
    glBegin(GL_POLYGON);
        for(it=points.begin();it!=points.end();it++){
            glTexCoord4fv(&it->x);
            glVertex4fv(&it->x);
        }//endfor
    glEnd();
}//end function
```

Beim Zeichnen der Fläche ist darauf zu achten, dass zusätzliche zu `glVertex..` auch `glTexCoord..` aufgerufen wird, um die Texturgrenzen anzugeben.

Einen Eindruck von den erzielbaren Effekten geben Abb. 13.5 und Abb. 13.6. Bei beiden Abbildungen handelt es sich um die Darstellung der gleichen Flächen, wobei die Faltung der zweiten Fläche allerdings erst in Abb.13.6 ersichtlich wird.

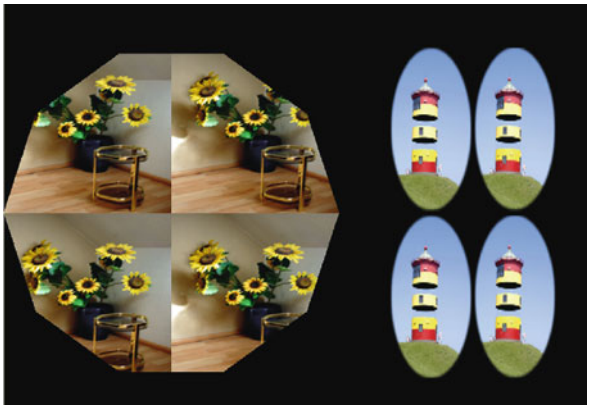


Abb. 13.5 Texturen in ebener Aufsicht

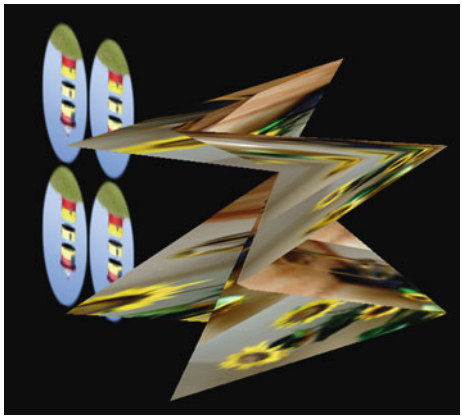


Abb. 3.6 Texturen auf gefalteter Fläche

13.7.2 Funktionen

Mathematisch beschreibbare Flächen liegen in der Form $z = f(x,y)$ oder – wesentlich häufiger – in der Form

$$P(s,t) = \begin{pmatrix} x(s,t) \\ y(s,t) \\ z(s,t) \end{pmatrix}$$

vor, hängen also von zwei Koordinaten ab. Wir können somit die `GridBuffer`-Klasse als Datenbasis für diese Objekte verwenden. Setzt man $x = s, y - t$, so gehen beide Formen ineinander über.

```
typedef
   Point const& (*SurfaceFunction)(GLfloat const&,
   GLfloat const&);
```

```
class SurfaceBuffer: public GridBuffer {
public:
    SurfaceBuffer();
    void finish();
    GLfloat smin, smax, tmin, tmax;
    int ns,nt;
    SurfaceFunction func;
};//end class
```

Die Methode zur Berechnung der Punkte wird als Funktionszeiger hinterlegt. Die Berechnung der Punkte erfolgt in der `finish()`-Methode, die in der `init()`-Methode des zugehörenden Objektes aufgerufen wird. Dieses ist auch für die Initialisierung des Funktionszeigers und der Attribute zuständig.

```
void SurfaceBuffer::finish(){
    GLfloat s(smin),t(tmin),
        ds((smax-smin)/(ns-1)),dt((tmax-tmin)/(nt-1));
    points.clear();
    set_dimension(ns,nt);
    for(int i=0;i<ns;i++,s+=ds){
        for(int j=0;j<nt;j++,t+=dt){
            set_grid_point(i,j,func(s,t));
        }//endfor
    }//endfor
    finish_done=true;
}//end function
```

13.7.3 Bezierflächen

Bezierflächen sind approximierte Flächen, bei denen die generierte Fläche außer durch die Randpunkte in der Regel nicht durch die Stützpunkte verläuft. Bezierflächen werden ebenfalls auf eine 2D-Gitter konstruiert. Bezüglich der Theorie sei auf die gängige Literatur verwiesen.

Die Klasse `Bezier_2D` wird von der Klasse `GridBuffer` abgeleitet und besitzt neben den überschriebenen Methode `draw()` zwei weitere Attribute für die Gittergröße der Zeichnung.

```
class Bezier_2D: public GridBuffer {
public:
    Bezier_2D();
    void draw();
    int nx,ny;
};//end class
```

Im Punktcontainer werden nur die Stützpunkte gespeichert; die zu zeichnenden Punkte werden von OpenGL direkt generiert und tauchen in der Anwendung nicht auf.

In der Initialisierungsphase werden die Daten an OpenGL übertragen und die notwendigen Berechnungen durchgeführt:

```
void Bezier_2D::draw(){
    glColor4fv(&color.r);
    glColorMaterial(GL_FRONT_AND_BACK,
                    GL_AMBIENT_AND_DIFFUSE);
    glColorMaterial(GL_FRONT_AND_BACK,GL_SPECULAR);
    glEnable(GL_COLOR_MATERIAL);
    glEnable(GL_MAP2_VERTEX_4);
    glEnable(GL_MAP2_NORMAL);
    glMap2f(GL_MAP2_VERTEX_4, 0.0 , 1.0, 4, dim1,
            0.0, 1.0, dim1*4, dim2, &points.front().x);
    glMapGrid2f(nx, 0.0, 1.0, ny, 0.0, 1.0);
    glEvalMesh2(GL_LINE,0,nx,0,ny);
    Color c=color;
    int m=mode;
    mode=GL_POINTS;
    glPointSize(3);
    color=Color(1,0,1,1);
    PointBuffer::draw();
    mode=m;
    color=c;
}// end function
```

Die genaue Bedeutung der Parameter entnehme man dem OpenGL-Handbuch. Die Zeichnung in der `draw()`-Methode fällt sehr kurz aus.

```
glEvalMesh2(GL_LINE,0,nx,0,ny);
```

Der Befehl überträgt die bereits fertig generierten Werte an das OpenGL-System.
Anmerkung 1. Zum Zeichnen von Kurven stellt OpenGL auch eine eindimensionale Bezier-Interpolationsmethode zur Verfügung.
Anmerkung 2. Man könnte vermuten, dass die Übertragung der Daten an OpenGL nur einmalig erfolgen müssen und die Vorbereitung daher in einer `finish()`-Methode untergebracht werden kann. Allerdings ist es dann nicht möglich, mehrere Bezierflächen innerhalb einer Szene unterzubringen, da keine Referenzen auf übertragene Daten vorgesehen sind. Aus diesem Grund wird alles in der `draw()`-Methode abgewickelt.[11]

[11] Es genügt sogar schon die Aktivierung einer anderen Spline-Methode (z.B. NURBS), um die Bezier-Parameter verschwinden zu lassen.

13.7.4 NURBS-Freiformflächen

NURBS sind ebenfalls interpolierende Splinefunktionen, die besser anpassbare Flächen liefern, theoretisch aber nicht unbedingt leicht zu verstehen sind. Für die Freude mit der Theorie sei dem Leser wie üblich auf das Studium eines Lehrbuches verwiesen. Beispiele für die unterschiedliche Flächendarstellung sind in Abb. 13.7 und Abb. 13.8 gegeben.

Die Definition der NURBS-Klasse ähnelt der der Bezier-Klasse

```
class NURBS_2D: public GridBuffer {
public:
    NURBS_2D();
    ~NURBS_2D();

    void draw();
    int k1,k2;
    virtual void make_knots(int dim, int k,
                            vector<GLfloat>& v);

protected:
    GLUnurbsObj* nurb;
    vector<GLfloat> knots1,knots2;
};//end class
```

Die Konstanten `k1` und `k2` enthalten die Ordnung der NURB-Splines, und die für die Darstellung notwendigen Knotengewichte werden mittels der Methode `make_knots(..)` berechnet und den beiden Vektoren `knots1` und `knots2` abgelegt (genaueres siehe Theorie) .

Für jedes NURBS-Objekt wird in glu ein Zeichenobjekt angelegt, das im Konstruktor erzeugt wird. Da der Umgang mit NURBS nicht ganz unproblematisch ist, kann eine Fehlerfunktion angeben werden, die bei einem Berechnungsfehler eine

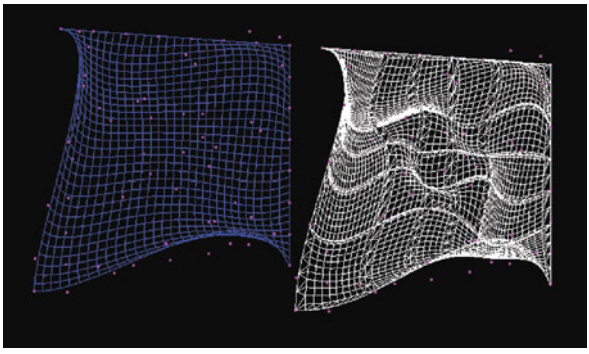


Abb. 3.7 Bezier (*blau*) und NURBS (*weiß*) Flächen mitdem gleichen Stützpunktsats

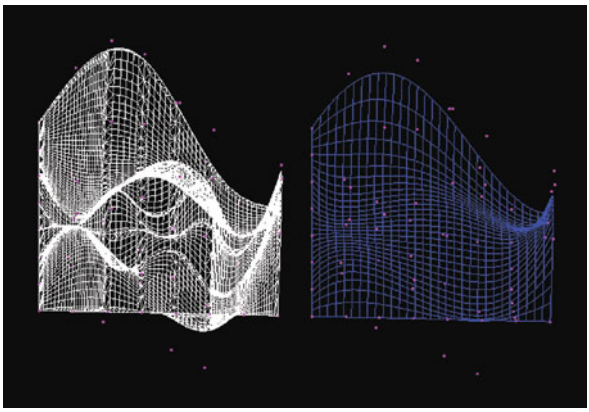


Abb. 13.8 Beizer (*blau*), NURBS (*weiß*)

Meldung ausgibt.[12] Die Berechnung der Splines und das Zeichnen der Fläche erfolgt ähnlich kompakt wie bei den Bezierkurve in der `draw()`-Methode.

```
NURBS_2D::NURBS_2D(){
    nurb=gluNewNurbsRenderer();
    gluNurbsCallback(nurb,
          GLU_ERROR,(void APIENTRY (*)()) nurbsError);
    k1=k2=4;
    mode=GLU_OUTLINE_POLYGON;
}//end constructor

NURBS_2D::~NURBS_2D(){
    gluDeleteNurbsRenderer(nurb);
}//end destructor

void NURBS_2D::make_knots(int dim, int k, vector<GLfloat>& v){
    v.resize(k+dim);
    for(int i=0;i<k+dim;i++){
        v[i]=max(0,min(i-k+1,dim-k+1));
    }//endfor
}//endfunction

void NURBS_2D::draw(){
    gluNurbsProperty(nurb, GLU_SAMPLING_TOLERANCE, 25.0);
    gluNurbsProperty(nurb, GLU_DISPLAY_MODE, mode);
```

[12] Hierbei kann es zu merkwürdigen Compilerfehlermeldung kommen, da die glu-Bibliothek ein anderes Aufrufschema verwendet als C++. Das Aufrufschema muss daher speziell angegeben werden.

```
    make_knots(dim1,k1,knots1);
    make_knots(dim2,k2,knots2);

    glColor4fv(&color.r);
    glColorMaterial(GL_FRONT_AND_BACK,
                    GL_AMBIENT_AND_DIFFUSE);
    glColorMaterial(GL_FRONT_AND_BACK,GL_SPECULAR);
    glEnable(GL_COLOR_MATERIAL);
    gluBeginSurface(nurb);
    gluNurbsSurface(nurb,knots1.size(),&knots1[0],
                    knots2.size(),&knots2[0],
                    4 , 4*dim1 , &points[0].x,
                    k1, k2, GL_MAP2_VERTEX_4);
    gluEndSurface(nurb);

    Color c=color;
    int m=mode;
    mode=GL_POINTS;
    glPointSize(3);
    color=Color(1,0,1,1);
    PointBuffer::draw();
    mode=m;
    color=c;
}//end function
```

13.8 Listenverwaltung durch OpenGL

OpenGL bietet die Möglichkeit, längere Befehlslisten einmalig zu speichern. Hierzu dienen die Befehle

```
glGenLists(..),
glNewList(...) ... glEndList()
glCallList(..)
```

Die erste Methode liefert einen freien Identifizierer für eine Liste, mit den beiden folgenden Befehlen werden alle zwischen diesen Befehlen ausgeführten OpenGL-Befehle wie Farben, Materialeigenschaften oder Punkte nebst ihren Zeichenmodi intern gespeichert. In der `display()`-Methode muss dann nur noch die Liste aufgerufen werden.

Da aufgrund des Listenaufrufs OpenGL-interne Werte verarbeitet und auf die im Programm gespeicherten Daten nicht mehr zugegriffen wird, eignen sich Listen nur für Daten, die sich während der Lebensdauer der Szene nicht ändern. Alle statischen Teile der Datenpuffer könnte man über Listen verwalten. Wir verzichten hier aber auf ein Beispiel.

13.9 Offene Probleme

Wir haben uns hier bemüht, die Berechnungsaufgaben soweit wie möglich an das Grafiksystem zu delegieren. Damit stoßen wir natürlich bei einer Reihe von Darstellungsaufgaben an die Grenzen. Hier einige Beispiele dazu:

(a) **Perspektivische Darstellung**. Das einfache Rechenmodell führt bei sehr breiter Objektsicht zu mehr oder weniger großen Verzerrungen. Statt der Z-Koordinate als einzige Abstandskoordinate müsste die echte Entfernung eines Objektes berücksichtigt werden, um auch Objekte am Bildrand unverzerrt darzustellen. Im anderen Extremfall können auch zusätzliche Verzerrungen gewünscht sein, die mit den Mitteln des OpenGL-Systems nicht implementierbar sind.
(b) **Spiegelung, Brechung**. Für komplexe fotografische Szenen ist das Materialbild zu simpel, da transparente Medien nicht dargestellt werden können. In der Folge sind auch Verzerrungen durch Brechung der Lichtstrahlen beim Durchgang durch transparente Medien nicht modellierbar. Spiegelungen von Objekten an reflektierenden Oberflächen lassen sich ebenfalls nicht modellieren.
(c) **Kollisionen**. In dynamischen Szenen ist häufig gefordert, dass Objekte einander nicht durchdringen, sondern aneinander abprallen. Unterliegen die Objekte unterschiedlichen Transformationen, so ist das aus den ursprünglichen Raumkoordinaten oft nicht abzulesen, sondern erst nach Durchführung der Transformation ermittelbar.
(d) ...

In diesen Fällen müssen viele Aufgaben, angefangen beispielsweise bei den Transformationen, außerhalb des OpenGL-Systems übernommen werden. Dies geht jedoch über den Rahmen dieser Einführung hinaus.

Die frei zugänglichen und nicht gerade vollständigen Attribute sowie die ungenügende Absicherung gegen Fehlprogrammierung (*siehe Anmerkungen in verschiedenen Kapiteln*) weist das Programmpaket ebenfalls als reines Übungspaket zum Eindringen in die OpenGL-Handhabung aus. Um tatsächlich zu einer für größere Projekte brauchbaren Klassenbibliothek zu werden, wäre insbesondere in diesem Bereich einiges zu tun. Das Spektrum der implementierten Datenklassen ist ebenfalls etwas dünn; bezüglich der Kombinationen Flächendarstellungen/Beleuchtungsparameter/Texturen besteht noch erheblicher Ergänzungsbedarf, zumal einige OpenGL-Möglichkeiten nicht berücksichtigt wurden.