D. Müller

Application Framework für die Entwicklung interaktiver Anwendungen mit multisensorischen Benutzungsschnittstellen

Ansätze einer prototypischen Realisierung von Rechnersystemen für die Produktion

artec - Arbeitspapier 22, April 1993

Inhalt

1. Einleitung

Mit dem vorliegendem Konzept wird versucht, einen Beitrag zum Problembereich der objektorientierten Softwareentwicklung für werkstattorientierte Softwaresysteme zu leisten.

Ausgangspunkt dieser Arbeit sind konzeptionelle Überlegungen zur erfahrungsorientierten Gestaltung von Rechnersystemen für die Produktion (vgl. Bruns 1993, Bruns/Heimbucher/Müller 1993). Im Rahmen dieser Forschung werden Anforderungen an werkstattorientierte Softwaresysteme entwickelt. Ziel des Ansatzes ist die Reduzierung rechnervermittelter zugunsten sinnlicher Wahrnehmung mit dem zentralen Aspekt der Werkstatt als Orientierungsrahmen.

Im Verlaufe der Arbeiten wurde sehr schnell klar, daß die praktische Umsetzung unserer Konzeption in Form von Prototypen notwendig sein würde. Nur so kann die Möglichkeit bestehen, neue Konzepte von benutzergerechten Systemen am Produkt *begreifbar* zur Diskussion zu stellen und so auch Nicht-Informatikern/-innen zugänglich zu machen.

Die vorliegende Darstellung beleuchtet schwerpunktmäßig software-technische Aspekte und soll als Ausgangspunkt für weitergehende Forschungen dienen.

2. Prototyp: Werkstatt-Layoutplanung mit Modellen

Mit der Entwicklung dieses Prototyps wurde der Versuch unternommen, neue vielsensorische Eingabetechniken zu implementieren und zu demonstrieren, wie die Modellierung geplanter technischer Anlagen mit einem materiellen Modell realisiert werden kann (vgl. Bruns 1993). Dabei kann ein reales Modell ("Klötzenwelt") per Hand aufgebaut, variiert und erweitert werden, parallel dazu können Analyseverfahren im Rechner ablaufen und wichtige Informationen nebenbei verarbeitet werden.

Die Benutzungsschnittstelle ist nicht der Rechner, sondern die funktional vorgefundene Welt. Der Rechner soll möglichst in den Hintergrund treten und nur dazu dienen, Informationen über die Objekte zu speichern (u. a. Form und Position). Eine einfache graphische Visualisierung der Szene kann zur Kontrolle einzelner Objekte herangezogen werden.

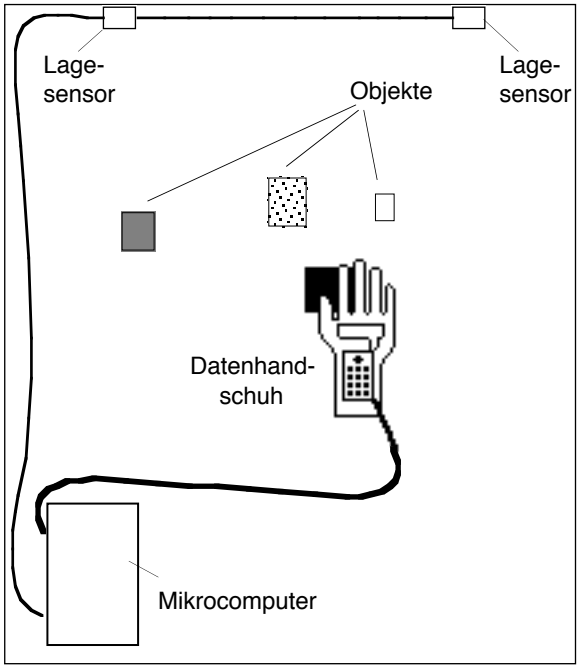

Abb. 2.1: Prototyp - Werkstatt-Layouterstellung mit Modellen (Draufsicht)

Als Eingabeschnittstelle wird ein Datenhandschuh benutzt; ein mit Sensoren ausgestatteter Handschuh, mit dem Daten über die Orts- und Lageposition der Hand sowie die Beugung der Finger an einen Rechner übermittelt werden können. Mit einer solchen sensorisierten Hand können reale Modelle (z. B. Holzmodelle) berührt, gegriffen und positioniert werden.

3. Objektorientierte Application Frameworks

Objektorientierte Programmierung ist besonders geeignet für die Realisierung wiederverwendbarer Software-Bausteine bzw. Klassen. Klassenbibliotheken bilden dementsprechend in der objektorientierten Systementwicklung die Grundlage der Konstruktion von Software. Bei den Klassen einer Klassenbibliothek kann zwischen sog. *Bausteinklassen* und *Frameworks* unterschieden werden.

Im Rahmen dieses Ansatzes sollen schwerpunktmäßig Frameworks als Design-Konzept für die Entwicklung werkstattorientierter Anwendungen betrachtet werden, um zu untersuchen, in wieweit hiermit wiederverwendbare und leicht erweiterbare Softwarekomponenten für *multisensorische* Eingabetechniken entwickelt werden können. In der objektorientierten Systementwicklung sind Frameworks ein Ersatz für Programmgerüste.

Das Grundprinzip eines Frameworks besteht darin, die Struktur oder das abstrakte Design für die Lösung einer bestimmten Problemstellung durch eine Menge von Klassen zu definieren (vgl. Gamma 1992, S 10ff). Im Gegensatz zu einem Programmgerüst kann ein Framework durch die Ableitung neuer Klassen wiederverwendet werden, ohne daß der Quellcode der Framework-Klassen geändert werden muß.

Eine spezielle Ausprägung von Frameworks sind die sog. *Application-Frameworks*. Sie stellen dem/r EntwicklerIn die Grundstruktur und das Gerüst einer vollständigen Applikation zur Verfügung. "An Application-Framework is a set of interconnected objects that provide the basic functionality of a working application, but which can be easily specialized (through subclassing and inheritence) into individual applications" (Schmucker 1986).

Da solche Frameworks die elementaren Funktionalitäten einer Applikation realisieren, werden sie auch als *generische* Applikationen bezeichnet.

Verfügbare Klassenbibliotheken und Application-Frameworks sind nur bedingt für die Umsetzung der oben skizzierten Aufgabenstellung geeignet, da sie die spezifischen Eigenschaften des zu erstellenden Prototypen nur unvollständig abzudecken in der Lage sind. Bibliotheken wie z. B. MacApp, ObjectWindows (OWL), THINK Class Library und ET++ unterstützen primär Benutzungsschnittstellen-Paradigmen, welche auf gängige Interaktionselemente wie Pop-Up bzw. Pull-Down-Menüs, Fenster, Rollbalken, Paletten, Dialogelemente usw. beruhen (zur Übersicht vgl. Valdés 1992). Die Architektur dieser Klassenbibliotheken berücksichtigt hauptsächlich konventionelle Eingabemedien wie Tastatur und Maus.

Bei der Familie von Applikationen, die von oben genannten Framework-Klassen unterstützt werden, handelt es sich um Anwendungen, die prinzipiell an Schreibtischtätigkeiten mit den charakteristischen Merkmalen der Textverarbeitung, des Archivierens, Kalkulierens usw. orientiert sind. Zentrales Objekt ist hier das (zwei-dimensionale) Dokument.

Anliegen unseres Konzeptes ist es aber, stärker werkstattorientierte Anwendungen zu betrachten. Es geht uns dabei um eine Rechnerunterstützung, die durch die Merkmale des Produkterzeugens, des Konstruierens, Gestaltens, Detaillierens, Fertigens charakterisiert ist. Zentrales Objekt ist hier das (drei-dimensionale) Werkstück in der Werkstatt. Elementare Operationen sind form-, funktions- und stoffbildende Tätigkeiten und Prozesse (vgl. Bruns 1993, Bruns / Heimbucher / Müller 1993).

Softwaretechnisch betrachtet, haben wir es in der Werkstatt nicht nur mit anderen Objekten zu tun, sondern es gilt auch, die Funktion des Rechners im Kontext einer *Werkstattperspektive* bis hin zur Systementwicklung zu reflektieren. Dies führt zu neuen Interaktionsformen mit Rechnersystemen. Übertragen wir beispielsweise das Konzept der *Direkten Manipulation* auf diesen Bereich, so geht es hier nicht um das Selektieren und Aktivieren von Ikonen auf dem Desktop, sondern um den direkten sinnlichen Umgang mit materiellen Objekten. Dieses weitgehende Konzept führt zu neuen Benutzungsoberflächen und neuen Funktionalitäten.

4. Implementierung

Bei der Entwicklung des beschriebenen Prototyps wurden folgende Prinzipien verfolgt:

1. Objektorientierung auf der Basis der Standardsprache C++
2. Portabilität und damit Verfügbarkeit auf verschiedenen Systemen (z. B. MS-DOS, Macintosh OS, MS Windows)
3. Übersichtlichkeit und Erweiterbarkeit.

Wir haben uns für eine Implementierung in C++ entschieden, da hierdurch einerseits vorhandene Interface-Routinen, die in ANSI C vorlagen, leicht integriert werden konnten (vgl. Jerchel 1992, Kraus 1992). Andererseits ist C++, trotz aller Mängel, relativ portabel, was beispielsweise für Object Pascal[1] nicht gilt.

Der vorliegende Prototyp basiert auf einem "Subset of C++", dabei wurde zum jetzigen Stadium aus Portabilitätsgründen bewußt auf einige fortgeschrittene Konzepte von C++, wie z. B. Mehrfachvererbung verzichtet.

Die ersten Implementierungsversuche wurden auf der Basis von Borland C++ Version 3.1 vorgenommen. Zunächst wurde dazu eine portierbare Grafik-Klasse entwickelt, die einerseits auf der Seite von MS DOS die BGI-Grafik von Borland und andererseits Apple Quickdraw unterstützt. Hierdurch ist es möglich, den Quelltext ohne Änderungen auf verschiedenen Zielmaschinen zu compilieren. Weitere Anpassungen an andere Systeme sind auf diese Weise leicht zu realisieren (vgl. Meyer 1992).

```
class CGraf: public CObject {
public:
   CGraf (void);
   virtual ~CGraf (void);
    virtual void  Show (void);
   virtual void  BackColor (int aCol);
   virtual void  ForeColor (int aCol);
...
   virtual void  Line (int x1, int y1, int x2, int y2);
   virtual void  LineRel (int dx, int dy);
   virtual void  MoveRel (int dx, int dy);
   virtual void  BeginDrawing (void);
   virtual void  EndDrawing (void);
   virtual void  Save (void);

protected:
  short itsWidth;
  short itsHeight;
  short itsMaxColor;
...
 };
```

Programm 4.1: Erste Ansätze einer portablen Grafikklasse (Klassendefinition)

[1] Ob die Entwicklung neuer Pascal-Dialekte, wie z. B. Pascal++ dieses Defizit reduziert, bleibt abzuwarten.

4.1. Objekte als Grundelemente der Implementierung

Charakteristisch für den objektorientierten Entwurf ist der Versuch, die *Welt* des jeweiligen Anwendungsfeldes im Computer in Form dezentraler Objekte zu rekonstruieren. Dabei geht man davon aus, daß ein solches Anwendungsfeld als Gesamtheit von Objekten (Dinge, Orte, Ereignisse usw.) aufgefaßt werden kann, in denen diese unterschiedlichen Objekte in einer gewissen Beziehungen zueinander stehen, sich im Zeitablauf ändern sowie wechselseitig beeinflussen können (Goldberg/Robson 1989, Meyer 1990, Booch 1991).

Übertragen auf den dargestellten Prototyp heißt das, daß wir eine *Modellwelt* definieren als einen quaderförmigen Aktionsraum, in dem einfache geometrische Objekte beliebig plaziert werden können. Jedes Teil dieser *Klötzchenwelt* (Hand, Aktionsraum, Einzelkörper) wird durch ein Anwendungsobjekt repräsentiert. Die Eigenschaften und Zustände sowie das Verhalten der Objekte (z. B. Hand greift, Körper wird plaziert) sind in Klassen definiert, die über eine Vererbungsstruktur allgemeine Eigenschaften zu speziellen verfeinert. Durch solche Klassenhierarchien erhält man redundanzarme Beschreibungen komplexer Konzepte, die erweiterbar sind: in weiteren Schritten kann den von einer Klötzchenwelt zu komplexeren Modellen übergegangen werden.

Die folgende Abbildung zeigt den ersten Entwurf einer vereinfachten Klassenhierarchie (Abb. 4.1). Dargestellt sind im Prinzip drei Arten von Klassen:

1. Klassen, in der die elementaren Objekte der zuvor beschriebenen Klötzchenwelt implementiert sind (World, Glove, 3D-Shape).

2. Die Applikations-Klasse (Application), die die prinzipielle Ablauf- und Interaktionsstruktur einer Anwendung beschreibt.

3. Anwenderklassen (My World, My Application), die das *spezielle* der jewei-ligen Anwendung definieren.

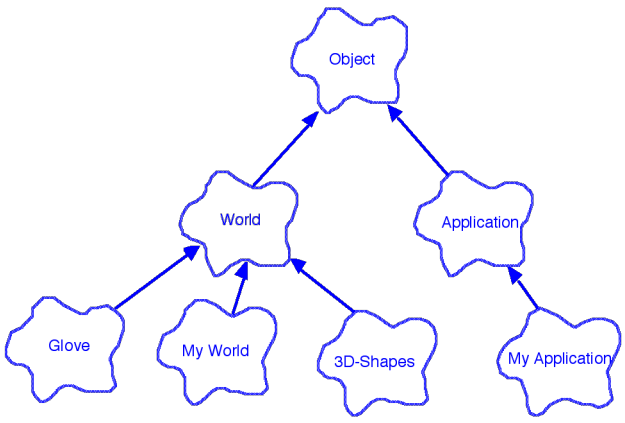

Abb. 4.1: Klassen - Diagramm

Die Zustände der jeweiligen Objekte werden durch Instanzenvariablen, ihre Funktionalität durch Methoden beschrieben.

Die einzelnen Objekte (wie Datenhandschuh, Körper im Aktionsraum usw.) interagieren durch Austausch von Meldungen miteinander. Ein geometrisches Objekt bekommt z. B. die Meldung *grab*, wenn es mit dem Datenhandschuh gegriffen wurde. Auf diese Weise lassen sich Aktionen modellieren, unabhängig davon, wie die jeweilige Anwendung im Einzelfall aussieht (vgl. Abb. 4.2).

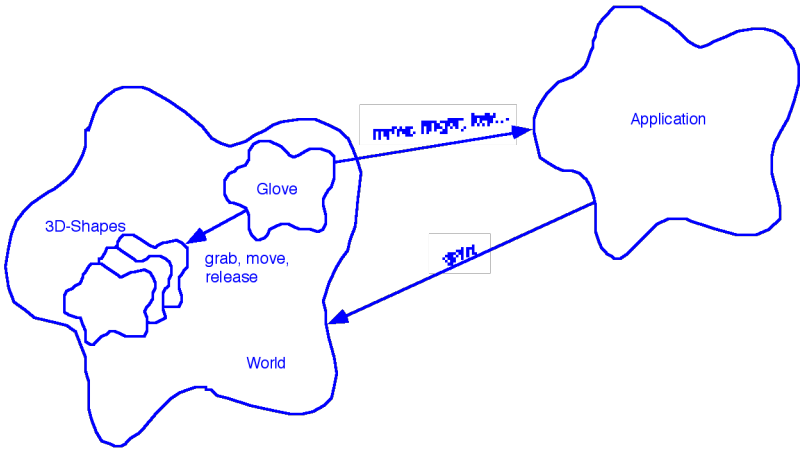

Abb. 4.2: Objekt - Diagramm

Eine kleine Liste von Meldungen, die vom Objekt Datenhandschuh an andere Objekte gesendet werden, möge die folgende Tabelle verdeutlichen.

	Meldung	**Aktion, Zustand**
1	Grab	Objekt wurde gegriffen
2	Release	Objekt wurde wieder losgelassen, es fällt zu Boden
3	Grabbed	Objekt wird festgehalten und bewegt
4	Move	Datenhandschuh ändert seine Position

Tabelle 4.1: Meldungen zwischen Objekten

4.2. Eingabeschnittstelle Daten-Handschuh

Mit neuen vielsensorischen Eingabemedien, wie etwa dem Datenhandschuh, besteht die Möglichkeit, reale Gegenständen in einem definierten Aktionsraum zu berühren, zu greifen oder zu bewegen; Information über Form und Lage der Objekte werden von einem Rechner quasi nebenbei protokolliert.

Seit einiger Zeit existieren auch Systeme, die eine recht einfache Anbindung eines Datenhandschuhs an einen Personal Computer ermöglichen (vgl. Jerchel 1992). Für unseren ersten *Gehversuche* stand uns der "PowerGlove" von Nintendo zur Verfügung.

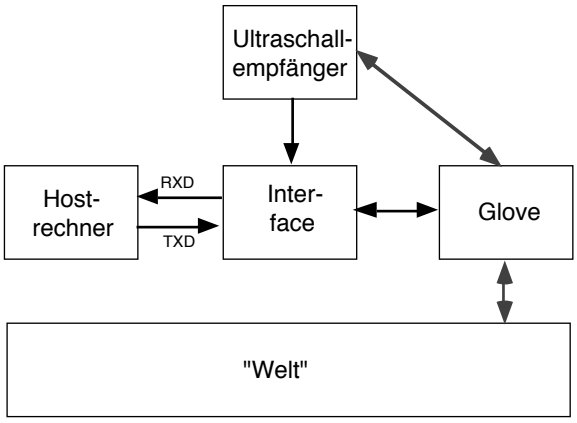

Abb. 4.3: Komponenten des PowerGlove-Systems

Der PowerGlove wurde in Zusammenarbeit mit Ingenieuren von AGE, VPL und Matell als vereinfachte Version des VPL-DataGlove speziell für Heim-Video-Spielsysteme von Nintendo konzipiert. Anstelle teurer Glasfaserkabel zur Bestimmung der Fingerkrümmung wird leitfähige Tinte eingesetzt, die auf spezielles Kunstoffmaterial aufgedruckt wird. Die Position der Hand im Raum wird per Ultraschall bestimmt. Hierzu werden aus drei Schallquellen Impulse ausgesandt und von einem Sensor im Handschuh empfangen. Dieser berechnet daraus die jeweilige Position im Raum. Diese Art der Messung ist bisweilen, z. B. bei Hindernissen, ungenau dafür aber außerordentlich preiswert.

Für vertiefende Arbeiten und komplexere Anwendungen ist ein höherer technischer Aufwand gewiß unumgänglich. Da die Entwicklung extrem schnell verläuft, wird sicherlich das Preis/Leistungsverhältnis dieser Systeme immer besser werden.

Eine Hardwareanpassung des Datenhandschuh - Interfaces war nur für den Apple Macintosh erforderlich. Für MS-DOS konnten wir auf gut dokumentierte und ausgetestete Routinen zurückgreifen (vgl. Jerchel 1992). Über eine transparente Schnittstelle erfolgt die Einbindung dieser Routinen. Auch hier dürfte eine Anpassung an andere Maschinen keine Probleme bereiten.

4.3. Ereignisbehandlung und Kontrollfluß

Interaktive Applikationen werden durch die Eingabe des Benutzers gesteuert. Aus diesem Grunde steht im Zentrum jeder Applikation die Behandlung von Eingabeereignissen (events). Diese Ereignisse können aus verschiedenen Quellen stammen. Dabei sollen *nicht* nur Tastatur und Maus, sondern möglichst eine große Vielfalt anderer Schnittstellen in unser Konzept einbeziehen.

Diese Überlegungen führten dazu, ein Ereigniskonzept für die Interaktion des Datenhandschuhs mit der Anwendung einzuführen und Ereignisse als das geeignete

Kommunikationsmittel zwischen Anwendungsobjekten und Schnittstellen-Routinen zu betrachten. Der Vorteil eines solchen Konzepts liegt außerdem darin, daß auch bestehende Klassenbibliotheken durch Überschreiben der Ereignis-Methoden leicht integriert werden können.

	Eingabemedium	Ereignisse
1	Tastatur	key down, key up
2	Maus	move (x,y), button down, button up
3	Joystick	move (x,y), key pressed
4	3D Maus	move (x,y,z), button down, button up
5	3D Spaceball	move (x,y,z), button down, button up
6	Datenhandschuh	move (x,y,z,rot, pitch, raw), finger-bend
7	Spracheingabe	sequence
8	Bilderkennung	new frame

Tabelle 4.2: Ereignisse

Die Verarbeitung von Eingabeereignissen erfolgt in mehreren Phasen, wie dies am Beispiel eines *Glove Events* zu erkennen ist.

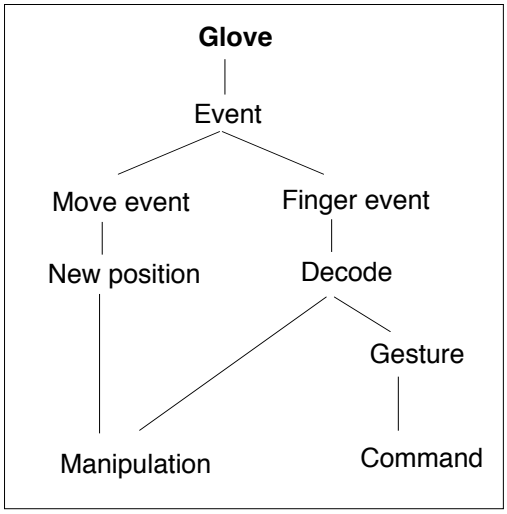

Abb. 4.3: Verarbeitung von Glove-Ereignissen

4.4. Ein Ansatz zur Gestik

Um dem Rechner Anweisungen zu geben, wie er mit den manipulierten Objekten verfahren soll, wurde ein Konzept von *Gesture-Commands* entwickelt.

Eine auf Gestik basierende Benutzungsschnittstelle eröffnet eine Alternative zu traditionellen Techniken (Tastatur, Menü usw.). Die Möglichkeit Objekte und Operationen mit einer einfachen intuitiven Geste zu spezifizieren, ist Voraussetzung dafür, mit realen Objekten direkter zu agieren. Tastatur und Maus scheiden dafür aus.

Ansätze, die diese Technik nutzen, beziehen sich meist auf den Bereich des Pen-Computing. Hier werden Gesten - gemeint sind handschriftliche Eingaben mit einem Digitalisierstift (hand markings) - verwendet, um verschiedene Programmfunktionen auszulösen: in der Regel für die Selektion, Manipulation und Archivierung von Textstellen und Graphiken.

Die Verwendung von Gestik im Kontext dieses Ansatzes muß als eine Interaktion auf der Basis einer in einem definierten Zeitlauf sich vollziehende Hand- und Fingerbewegung zum Zwecke der Kommando-Eingabe betrachtet werden (vgl. Rubine 1991).

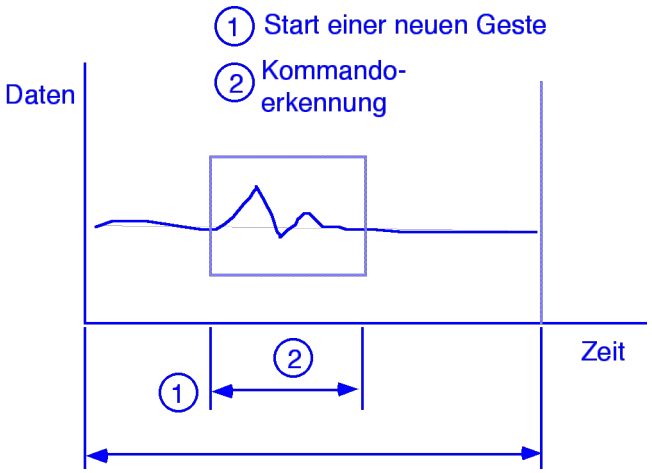

Abb. 4.6: Gestik als zeitbasierter Datentyp

Software-technisch haben wir es mit einem zeitbasierten Datentyp zu tun. Damit taucht das Problem auf, wie Beginn und Ende einer Geste erkannt wird.

Im Rahmen des vorhanden Prototyps wurde zunächst eine einfache Strategie auf der Basis einer statischen Gestik implementiert. Bessere Ergebnisse ergibt ein Verfahren, das ein gleitendes Fenster über die Folge von gespeicherten Daten (z.B. Datenstrom des Datenhandschuhs) verschiebt und eine Korrelationsanalyse durchführt. Liegt ausreichende Übereinstimmung dieser Daten mit einem vorgebenen Muster vor, wird dies als eine Geste interpretiert (vgl. Abb. 4.6).

5. Struktur und Inhalt einer Klassenbibliothek

Im folgenden soll die Struktur einer *Klassenbibliothek* dargestellt werden. Sie basiert auf praktische Erfahrungen mit verschiedenen Klassenbibliotheken, wie MacApp und der THINK Class Library (beide Implementierungen für den Apple Macintosh) und berücksichtigt die spezifischen Anforderungen von Applikationen, bei denen ein Datenhandschuh als Eingabemedium eingesetzt wird.

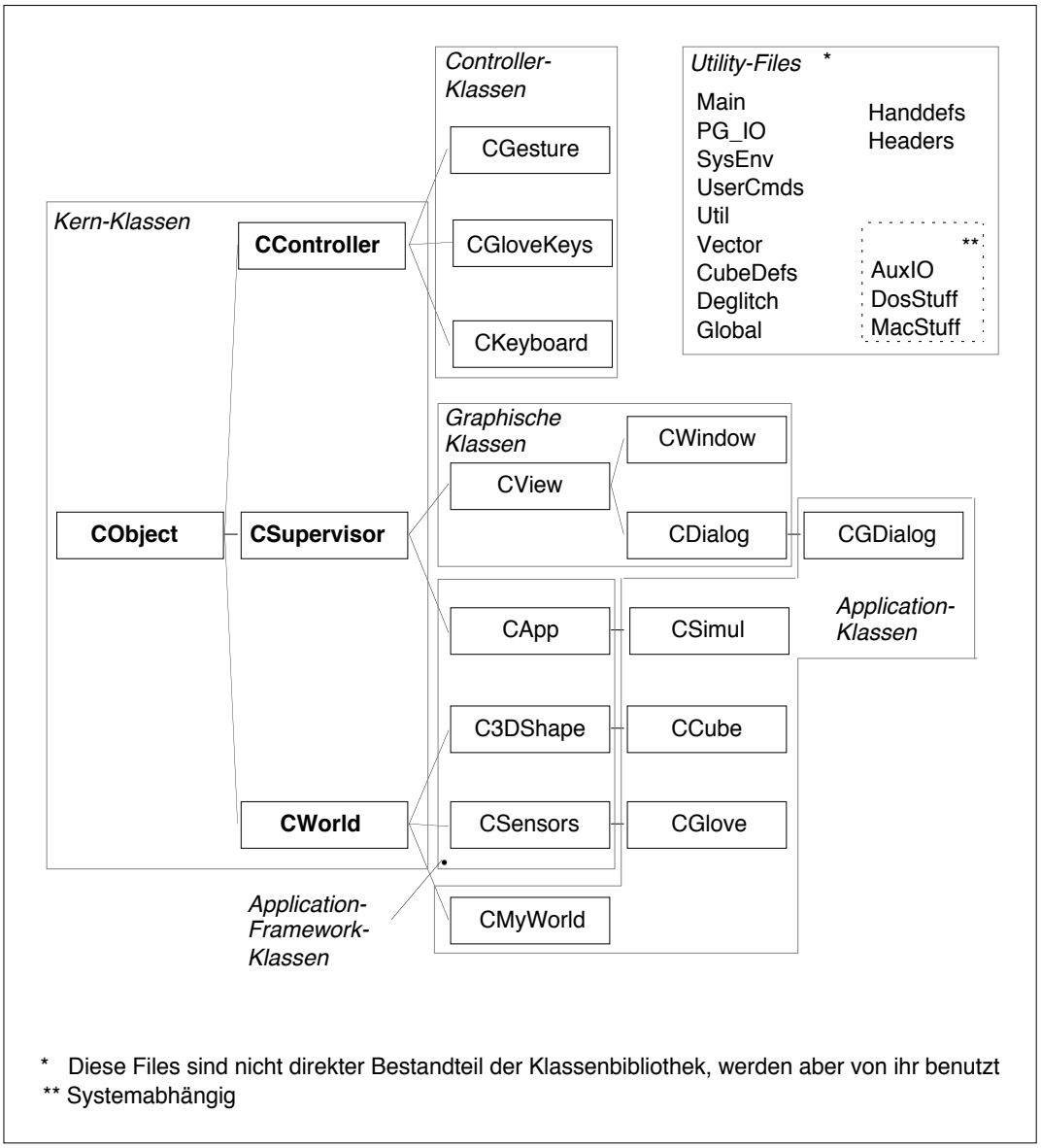

Abb. 5.1: Struktur des Application-Frameworks und Klassenhierarchie

Der damit erstellte Prototyp ergibt ein lauffähige Version, beschränkt sich aber auf elementare Funktionalitäten.

Um die Architektur dieser Bibliothek zu verstehen, ist es hilfreich, zunächst einmal ihre Struktur sowie das Zusammenspiel einzelner Objekte zu betrachten.

Die Abbildung 5.1 zeigt die Klassenübersicht. Die zentralen Klassen sind neben CWorld, CSupervisior and CController die daraus abgeleiteten Klassen C3DShape, CCube, CGlove, CWindow, CApp, CSimul und CGesture. CWorld, CSupervisior und CController bilden die Grundstruktur der Bibliothek und sind abstrakte Klassen[2] , die überschrieben werden müssen. Sie enthalten jeweils die grundsätzlichen Funktionalitäten bzw. Methoden.

Soweit für das konzeptionelle Verständnis notwendig, erfolgt nun eine Beschreibung der wichtigsten Klassen und Methoden im einzelnen.

5.1. Kernklassen

Die Kernklassen CObject, CWorld, CSupervisior and CController stellen die Infrastruktur für die Klassenbibliothek zur Verfügung. Sie befinden sich auf der konzeptionell untersten Schicht, d. h. alle anderen Klassen sind von ihnen abgeleitet. Die Kernklassen bestehen hauptsächlich aus virtuellen Methoden, die in den abgeleiteten Klassen implentiert werden müssen.

Klasse CObjekt

CObject ist eine sog. Root-Level-Class, von ihr sind alle anderen Klassen abgeleitet. In CObject sind Methoden zur Instanzierung und Löschung von Objekten implementiert. Diese Methoden werden von allen anderen Klassen des Application Frameworks benutzt.

Klasse CController

CController ist eine abstrakte Klasse, die die Interaktion des Benutzers mit der Applikation definiert. Für jedes Eingabemedium wird eine entsprechende Unterklasse implementiert und die Methode Decode überschrieben. Decode versucht Ereignisse (Events) in Commands zu übersetzen. Falls ein gültiges Kommando vorliegt, sendet diese Methode eine DoCommand-Message.

```
class CController : public CObject {
public:
  CController (void);
  virtual ~CController (void);
  virtual Boolean Decode (void);
};
```

Programm 5.1: Klassendefinition CController

[2] Abstrakte Klassen dienen dazu, um auf der oberster Schicht allgemeine Programmstrukturen zu definieren, sie enthalten keine Implementation einzelner Methoden. Von abstrakten Klassen können deshalb auch keine Instanzen gebildet werden.

Am Beispiel der Verarbeitung von Gesten-Komandos soll die Funktionalität der Controller-Methoden insbesondere DoCommand- bzw. Decode-Methode illustriert werden (vgl. hierzu auch Kap. 5.5):

Für den korrekten Austausch von Befehlen (Commands) existiert eine spezielle Variable mit dem Namen `gReceiver`, die immer immer auf dasjenige Objekt zeigt, das als erstes ein Kommando empfangen und bearbeiten soll. Der/die ProgrammiererIn ist dafür zuständig, jeweils diese Variable mit der Methode `SetReceiver` zu setzen. Alle von der Klasse CSupervisor abgeleiteten Objekte, sind in der Lage, Commands zu empfangen und durch die Methode DoCommand auszuführen.

Innerhalb der Methoden `DoCommand` werden die jeweiligen Commands analysiert, gebenenfalls ausgeführt oder in der Vererbungshierarchie durchgereicht. Folgende *command-chain* verdeutlicht diesen Vorgang:

1. Glove erzeugt ein Ereignis (z. B. finger event)
2. Das Glove Objekt sendet daraufhin dem entsprechendem Controller Objekt (z. B. CGesture) eine Meldung (Save).
3. Das Controller -Objekt versucht diese Meldung zu dekodieren (Decode), falls ein Kommando vorliegt (z. B. Geste), sendet das Controller-Objekt eine DoCommand Nachricht.
4. Die `DoCommand` Nachricht wird von demjenigen Objekt empfangen, welches durch die Methode `SetReceiver` als letztes gesetzt worden ist. Dies ist ein Objekt einer View oder Application Klasse.
5. Das Empfänger-Objekt analysiert das Kommando und führt es aus oder reicht es an seine Oberklasse weiter

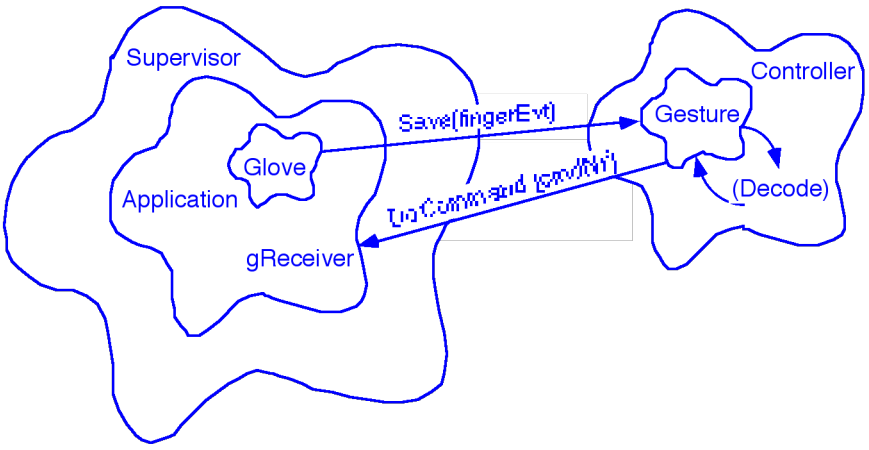

Abb. 5.2: Gesten-Kommando-Verarbeitung

Klasse CSupervisor

Die Klasse CSupervisor ist ebenfalls eine abstrakte Klasse, welche die Kommunikation zwischen von ihr abgeleiteten Klassen und den Controller-Klassen über einen Command-

Mechanismus steuert. Commands (Befehle) lösen bestimmte Funktionen aus, sie werden von Controller-Objekten gesendet.

Von CSupervisor abgeleitete Klassen müssen die DoCommand-Methode überschreiben, um auf Commands zu reagieren. Mit der SetReceiver-Methode kann die Interaktion zwischen dem Absender eines Commands und dem Empfänger gesteuert werden.

```
class CSupervisor : public CObject {
public:
  CSupervisor (void);
  virtual ~CSupervisor (void);

  virtual void SetReceiver (CSupervisor *aReceiver);
  virtual void DoCommand (tCommand nr);
};
```

Programm 5.2: Klassendefinition CSupervisor

Klasse CWorld

CWorld beinhaltet die grundsätzlichen Methoden zu Instanzierung und Darstellung eines Weltmodells, also körperhafter Objekte, die Form und Lokalität besitzen. *CWorld* ist eine abstrakte Klasse für alle Weltobjekte. Neue Objekteigenschaften müssen in abgeleiteten Klassen definiert werden.

```
class CWorld : public CObject {
public:
  CWorld (void);
  virtual ~CWorld (void);

  virtual void  Draw (void);
  virtual void  SetWorld (tWorld *w);
  virtual void  GetWorld (tWorld *w);

  virtual void  SetScale (int xScale, int yScale);
  ...
  virtual void    Show (Boolean b);
  virtual Boolean Visible (void);
   ...
};
```

Programm 5.3: Klassendefinition CWorld

5.2. Application-Framework-Klassen

In dieser Gruppe sind diejenigen Klassen angeordnet, die das abstrakte Modell einer Applikation definieren. Die Klasse CApp bildet das Grundgerüst für eine Anwendung. Sie verwaltet den globalen Zustand jeder Applikation. Beim Starten einer Applikation wird genau eine Instanz dieser Klasse bzw. der von ihr abgeleiteten Klasse erzeugt. In der globalen Variable gApplication wird dieses Objekt gespeichert.

Klasse CApp

CApp ist eine abstrakte Klassen für eine Applikation mit multisensorischen Eingabemedien. Für die jeweilige Anwendung muß diese Klasse überschrieben werden. CApp definiert die generelle Struktur des Programms und die Interaktion mit den Eingabemedien (vgl. Programm 5.4 und 5.5).

Das Programm läuft in einer Endlosschleife bis es vom Benutzer abgebrochen wird. In dieser Schleife werden alle auftretenden Ereignisse (events) bearbeitet, sie ist in der Methode *Run* definiert. Bei jedem Schleifendurchlauf werden folgenden Methoden ausgeführt:

- ProcessWorldEvents,
- ProcessSysEvents,
- PerformWorldTasks und
- UpdateWorld.

ProcessWorldEvents ist als öffentliche (public) Methode in CApp definiert und muß in der entsprechenden Unterklasse, z. B. *CSimul*, überschrieben werden. In dieser Methode wird die Kommunikation mit dem DataGlove bzw. der vorhandenen Eingabeschnittstelle programmiert.

```
class CApp : public CSupervisor {
public:
  CApp (void);
  virtual ~CApp (void);
  void Run (void);
  virtual void Exit (void);

  virtual void ProcessWorldEvents (void);
  virtual void ProcessKeyEvents (char key);
  virtual void ProcessSysEvents (void);

  virtual void PollGlove (void);
  virtual CGlove *GetGlove (void);

  virtual void PerformWorldTasks (void);
  virtual void UpdateWorld (void);
  virtual void DoCommand (tCommand nr);
   ...
protected:
  CGlove   *itsGlove;
};
```

Programm 5.4: Klassendefinition CApp

ProcessSysEvents ist eine Methode, in welcher die Ereignisbehandlung des jeweiligen Rechner beschrieben werden kann (Keyboard, Windows, Mouse usw.).

PerformWorldTasks ist eine Methode, mit der autonome Aktivitäten von einzelnen Objekten definiert werden können. Dies könnten Tasks sein, die automatisch im Hintergrund ablaufen sollen (z. B. Simulation von Fallbewegungen oder Statusanzeigen).

Die Methode UpdateWorld dient dazu, die Darstellung des Modells zu aktualisieren.

Die *DoCommand*-Methode bearbeitet Benutzer-Befehle (Commands). In dieser Klasse ist das Standardverhalten auf Commands definiert. In abgeleiteten Klassen sollte DoCommand überschrieben werden, um spezifischen Befehle zu bearbeiten. Commands,

die in diesen Klasse nicht bearbeitet werden können, sollten zur nächsten Klasse durchgereicht werden (command-chain).

```
// ------------------------------------------------------
// CApp ()
//   Init application
CApp::CApp (void)
{
   InitMachine(5);
   SetUpMenus (1);
  ...
   SetReceiver (this);
}
...

// ------------------------------------------------------
// Run ()
//   This is the >>main event loop<<
//   Run the application by processing events.
void CApp::Run(void)
{
   do {
      ProcessSysEvents ();
      ProcessWorldEvents ();
      PerformWorldTasks ();
      if (gSysTime > itsUpdateTime)
         UpdateWorld ();
   } while (gUserQuit == FALSE);
}

// ------------------------------------------------------
// ProcessWorldEvents ()
//   Handle glove events, sensors etc.
//   this method must be overridden
void CApp::ProcessWorldEvents(void)
{
     // Subclass responsibility
}

// ------------------------------------------------------
// ProcessKeyEvents ()
//   If we use the keyboard (sorry), define a subclass method
//   Default action is to do nothing
void CApp::ProcessKeyEvents (char key)
{
     // Subclass responsibility
}
...

// ------------------------------------------------------
// PerformWorldTasks ()
//   Allow autonomous activities for objects
//   this method can be overridden
void CApp::PerformWorldTasks(void)
{
     // Subclass responsibility
}

// ------------------------------------------------------
// UpdateWorld ()
//   Update world: positions, display etc.
//   A subclass must define a new UpdateWorld method
//   and must call this method too, to maintain time update
void CApp::UpdateWorld (void)
```

```
{
   itsUpdateTime = gSysTime + kUpdateDelay;
}

// ------------------------------------------------------
// DoCommand ()
//   Define a subclass method for proccessing commands
//   Default action is to do some standard actions
void CApp::DoCommand (tCommand nr)
{
   switch (nr) {
      case cmdQuit:
         gUserQuit = TRUE;
         break;
      ...
   }
   itsLastCmd = nr;
}
```

Programm 5.5: Implementierung CApp

Klasse C3DShape

C3DShape stellt eine Klasse für einfache grafische Weltobjekte (3D-Körper) dar (vgl. Programm 5.6). In dieser Klasse sind Methoden zur Positionierung und zur Manipulation (Greifen, Loslassen) definiert. Eine wichtige Methode ist hier SetGlove, die jedem grafischen Objekt *seinen* Glove zuordnet.

```
class C3DShape : public CWorld {

public:
   t3DShape  itsShape;    // new position, speed etc.
   CGlove    *itsGlove;   // active glove
    ...
public:
  C3DShape (void);
  virtual ~C3DShape (void);
  void SetNew (t3DShape *aPos);
  void SetPos (int x, int y, int z, int r);
  void GetPos (t3DShape *aPos);
  void SetSpeed (int vx, int vy, int vz, int vr);

  // grabbing
  void Grab (void);
  void Release (void);
  Boolean  Grabbed (void);

  // give him a glove
  virtual void SetGlove (CGlove *aPos);

  // drawing
  virtual void Draw (void);

  // world
  virtual Boolean WorldMax (VECTOR  *ncube, t3DShape *aCube);
  virtual int  CalcMax (void);

  // misc
  virtual void BetweenFingers (void);
```

```
  virtual long Distance (VECTOR *vCube, VECTOR *vGlove, long
*dir);
};
```

Programm 5.6: Klassendefinition CCube

Klasse CSensors

CSensors ist eine abstrakte Klasse für Eingabemedien (Input-/Output Devices). Spezifische Funktionalitäten für bestimmte Eingabetechniken müssen in abgeleiteten Klassen implementiert werden (vgl. Programm 5.7).

```
class CSensors: public CWorld {

public:
  CSensors (void);
  virtual ~CSensors (void);
  virtual void Reset (void);

  // test stuff
  virtual tError  ErrStatus (void);
  void SetErrStatus (tError err);

private:
   tError  itsErrStat;
protected:
   int   itsErrCount;
};
```

Programm 5.7: Klassendefinition CSensors

5.3. Applikation-Klassen

Zu dieser Gruppe von Klassen gehören all diejenigen, die das Verhalten und die Eigenschaft einer spezifischen Applikation beschreiben. Sie müssen für jede Anwendung neu implementiert werden. U. a. werden hier die virtuellen Methoden der Application-Framework-Klassen durch konkrete Methoden überschrieben.

Klasse CSimul

In der Klasse CSimul sind diejenigen Methoden implementiert (vgl. Programm 5.8/5.9), die in der Oberklasse CApp abstrakt definiert sind. Im wesentlichen sind dies neben Konstuktor und Destruktor folgende Methoden: ProcessWorldEvents, ProcessKeyEvents, Manipulate3DShape, New3DShape und Find3DShape.

ProcessWorldEvents implementiert die spezifischen Ereignismethoden für das jeweilige Eingabemedien, in diesem Fall den Datenhandschuh. Immer dann, wenn dieser Datenhandschuh seine Positionswerte ändert, wird ein Ereignis (event) ausgelöst. Ist ein solches Ereignis aufgetreten, wird durch Find3DShape geprüft, ob die sensitive Stelle eines Objektes berührt wurde, wenn ja, wird die gewünschte Aktion durch die Methode Manipulate3DShape ausgeführt.

Bei jedem Schleifendurchlauf werden Objektmodell und Handpositionswerte aktualisiert und gespeichert.

Die Methode New3DShape dient zur Instanziierung eines neuen 3D-Objektes im Aktionsraum, in unserem Fall ein Würfel.

```
class CSimul : public CApp {
public:
  CSimul (void);
  virtual ~CSimul (void);
  virtual XSimul (void);

  void ProcessWorldEvents (void);
  void PerformWorldTasks (void);
  void ProcessKeyEvents (char key);
  void UpdateWorld (void);
  void DoCommand (tCommand nr);

private:
  CCube     *itsCube[kMaxCube];
  C3DShape  *itsActiveShape;
  CMyWorld  *itsWorld;
   ...
  void Manipulate3DShape(void);

  void New3DShape (int n);
  void Find3DShape (void);
  ...
};
```

Programm 5.8: Klassendefinition CSimul

```
...

// ------------------------------------------------------
// ProcessWorldEvents
// This is called form the main event loop
//   Read glove event and handle it
void CSimul::ProcessWorldEvents(void)
{
   tGloveEvent   event;
   event = itsGlove->GloveEvent ();
   if (event != nullEvt) {
      Find3DShape();
      Manipulate3DShape();
   }
}

// ------------------------------------------------------
// Manipulate3DShape
//  Manipulate 3d-object
void CSimul::Manipulate3DShape(void)
{
   // get old and new position of glove
   // calculate diff between new and old position
   // move glove in small steps to recognize every collision
   // with other objects
   // check, if cube is manipulated by glove
}

// ------------------------------------------------------
```

```
//
void CSimul::New3DShape (int n)
{
}

// -----------------------------------------------------
// Find3DShape
//  look for an object
void CSimul::Find3DShape (void)
{
}
```

Programm 5.9: Implementierung CSimul

Klasse CGlove

CGlove ist eine abgeleitete Klasse von *CSensors,* sie enthält Methoden zur Orientierung und Positionierung des angeschlossenen Datenhandschuhs vgl. Programm 5.10). Hier werden die jeweiligen Interface-Routinen aufgerufen und alle Handpositionswerte verwaltet. Zentral ist die Methode *GloveEvent*: sie meldet anderen Objekten die Ereignisse die vom Datenhandschuh (Glove Events) ausgelöst werden. Normalerweise wird zur Laufzeit des Programms, nur ein einziges *Glove-Objekt* erzeugt, nämlich für jeden Datenhandschuh eines. Bei der Verwendung von mehreren Handschuhen, werden auch mehrere Glove-Objekte instanziiert.

```
class CGlove : public CSensors {
  tGlove   itsGlove;
  tGlove   itsOldGlove;

public:
  CGlove (void);
  virtual ~CGlove (void);
  ...
  // change pos
  void RotateY (void);
  void SetNew (tGlove *aGlove);

  // access
  Boolean GloveReady (void);
  tGloveEvent GloveEvent (void);
  ...
  Boolean AllBend (void);

  // noise removal
  void Deglitch (void);
private:
   // io
  void IORead (void);
  ...
};
```

Programm 5.10: Klassendefinition CGlove

5.4. Graphische Klassen

Die graphischen Klassen implementieren die Eigenschaften und Funktionalitäten der visuellen Elemente einer Anwendung. Generische Eigenschaften und Methoden sind in CView, speziellere in Windows- und Dialog-Klassen definiert.

Klasse CView

CView ist die zentrale abstrakte Klasse der graphischen View-Klassen. Mit *Views* werden hier Objekte verstanden, die in irgendeiner Form visuell repräsentiert sind bzw. den Kontext für die (grafische) Visualisierung des Modells zur Verfügung stellen (vgl. Programm 5.11*)*.

Im einfachsten Fall ist dies der gesamte Bildschirm oder ein Ausschnitt desselben mit einer festgelegten Größe. Jedes View-Objekt besitzt einen eigenen grafischen Kontext oder *Port*. Ein Port ist charakterisiert durch die Eigenschaften des grafischen Kontextes (Größe, Farbe usw.).

```
class CView : public CSupervisor {

public:
  CView (void);
  virtual ~CView (void);

  virtual void Show (void);
  virtual void Hide (void);
  Boolean Visible (void);

  void SetSize (int width, int height);
  int GetMaxX (void);
  int GetMaxY (void);

};
```

Programm 5.11: Klassendefinition CView

Klasse CWindow

CWindow dient dazu, ein Fensterobjekt zur Darstellung der grafischen Repräsentation des Modelles zur Verfügung zu stellen. Fenster im klassischen Sinne - gemeint sind beispielsweise mehrere überlappende Windows - werden in dieser Anwendung nicht benötigt, sie sind sogar unnötig und würden unser konzeptionelles Ansinnen verwässern, nämlich nur das notwendige auf dem Rechnerbildschirm abzubilden.

5.5. Controller-Klassen

Die Controller-Klassen implementieren die verschiedenen Methoden zur Interaktion des Benutzers mit der Anwendung. Für jedes Eingabemedium (Tastastur, Datenhandschuh usw.) muß eine entsprechende Unterklasse implementiert werden. Durch dieses Konzept, werden ähnlich wie in SmallTalk, alle Bedienungselemente von der Modellkomponente getrennt.

Klasse CGesture

In dieser Klasse ist die Bearbeitung von *Gesture-Commands* implementiert (vgl. Programm 5.12*)*. Die Methode Decode versucht eine Gestik zu entschlüsseln, indem der dynamische Ablauf einer Geste untersucht wird. Mit der Methode Change können neue Gesten definiert werden. Hierdurch ist es möglich, verschiedene Gesten zu speichern, dies ist besonders wichtig, um Handpositionen und Fingerkrümmungen bestimmten Gestiken zuzuordnen, z. B. Greifen, Loslassen, Berühren usw.. Soll die Strategie der Gesten-Erkennung erweitert werden, so kann dies durch Ändern oder überschreiben der Decode-Methode erfolgen.

```
enum nGesture {
   gestNull = -1,
   gestGrasp = 0,
   gestRelease,
   gestShow,
   ...
};

class CGesture : public CController {
   ...
  CGlove    *itsGlove;
public:
  CGesture (void);
  virtual ~CGesture (void);

  void SetGlove (CGlove *aGlove);

  virtual void Save (tsChar aByte);
  virtual Boolean Decode (void);
  virtual void Change(void);
};
```

Programm 5.12: Klassendefinition CGesture

6. Weitere Arbeiten, Ideen und Forschungsfragen

• *Erweiterung des Application-Frameworks:*

Es ist sinnvoll einen sogenannten *Change-Propagation-Mechanismus* zu implementieren. Dieser Mechanismus erlaubt es einem Objekt, Änderungen eines anderen Objekts zu beobachten.

Das Udaten der Szene (Neuzeichnen aller 3D-Objekte) könnte außerdem durch eine Dependency - Strategie, ähnlich wie in Smalltalk erfolgen. Prinzip: In einer Root-Klasse wird eine Liste aller darzustellenden Objekte verwaltet. Über einen Trigger-Mechanismus in der Main-Event-Loop kann diese Liste mit einer DoForEach-Methode durchlaufen und alle Objekte aus dieser Liste können damit neu gezeichnet werden. In Verbindung mit einer Change-Propagation wäre dies automatisierbar.

• *Domänen Analyse*

Bezogen auf die Entwicklung eines Application-Frameworks für werkstattorientierte Systeme müßte detaillierter als es in diesem Zusammenhang möglich ist, eine objektorientierte Domänen Analyse (*Domain Analysis*) durchgeführt werden. Ziel einer solchen Analyse wäre es, die gemeinsamen Eigenschaften einer Familie von Applikationen zu erkennen, um daraus ein entsprechende Struktur für den Entwurf eines Frameworks zu entwickeln (vgl. Booch 1991, S 142 ff).

• *Verbesserung der Portabilität:*

Das Konzept einer portablen Ein- und Ausgabeschnittstelle könnte in der Richtung verbessert werden, indem eine oder mehrere abstrakte Schnittstellen-Klassen (sog. Brücken- oder Implementationsklassen) Ein- und Ausgabeoperationen definieren (vgl. Geary 1990, Gamma 1992, S. 123). Spezielle Klassen konkretisieren diese abstrakten Klassen jeweils für spezielle Hardwareanbindungen. Dazu bedarf es eines generellen Konzeptes von I/O-Operationen, die für jegliche Art von Interfacetechniken tragfähig ist.

• *Einbindung eines CAD-Systems:*

Es ist lohnend zu untersuchen, ob und inwieweit eine 3D-Grafikbibliothek, CAD-Toolbox oder gar ein ganzes CAD-Systeme integriert bzw. genutzt werden könnte. Hierdurch wäre es möglich, beispielsweise folgende Funktionalitäten ohne zusätzlichen Programmieraufwand zu implementieren:

1. Generierung von 3D-Objekten (Grundkörper als Drahtmodell)
2. Modellierung komplexer zusammengesetzter Objekte
3. Modifizierung von Objekte bezüglich Form und Abmessungen
4. Bewegung von Objekten
5. Verwaltung geometrischer Daten

Im Hinblick auf diese Anforderungen müßte ein solches System eine Programmier-Schnittstelle, möglichst objektorientiert, zur Verfügung stellen.

• *Schnittstelle zu anderen Applikationen:*

Es sollte untersucht werden, inwieweit die Funktionalität anderer Applikationen und/oder Prozesse genutzt werden kann. Auch ist ein dynamischen Datenaustausch zwischen verschiedenen Rechnern eine gängige Problemstellung, bei der es gilt, reale Prozesse, wie z. B. Maschinensteuerungen anzukoppeln. Eine Schwierigkeit hierbei ist, daß die verschiedenen Betriebssysteme, sehr unterschiedliche Konzepte anbieten, deshalb müßte auch hier über ein generalisierbaren Ansatz nachgedacht werden.

• *Systematisierung der Interaktion mit realen Objekten (Interaction-Tasks):*

Multisensorielle Eingabeschnittstellen müßten noch genauer im Hinblick auf ihre Eignung als für die Interaktion mit *realen* Objekten untersucht werden. Dazu sollte über eine Systematisierung von Interaktionsklassen nachgedacht werden. Folgende Kategorien lassen sich unterscheiden:

- Navigation
- Selektion
- Interaktion
- Modifikation
- Kommandos

• *Formverändernde und nicht verändernde Interaktion und deren Implementation*

Zu untersuchen wären auch verschiedene Paradigmen zur formgebenden/-verändernden und *nicht* verändernden Interaktion bzw. Manipulation, wie bespielsweise:

- Bewegen (moving),
- Berühren (touching),
- Stoßen (pushing),
- Greifen (grabbing/release),
- Kneten (kneading) und
- Formen (shaping).

Dazu könnte eine *Taxonomie der Manipulation* entwickelt werden (vgl. Schultz 1992, pp. 24).

• *Ausbau des Konzeptes der Gesture-Commands:*

Um dem Rechner Anweisungen zu geben, wie er mit den manipulierten Objekten verfahren soll, wurde von uns ein Konzept von Gesture-Commands entwickelt. Dieser Ansatz muß noch weiter verfeinert werden: dazu sollte u.a. das dynamische Zeitverhalten von Gestik genauer untersucht und eine entsprechende Datenabstraktion implementiert werden (vgl. Rubine 1991).

Zu differenzieren wäre sinnvollerweise zwischen einer Geste (*gesture*) und einer stationären Handposition (*posture*) (vgl. Kaufmann/Yagel/Bakalash 1990).

7. Literatur

Booch, G. (1991): Object oriented design with applications. Redwood, Cal.

Borland GmbH (Hrsg.)(1992): Borland C++. Programmierhandbuch. Vers. 3.1. Starnberg.

Bruns, W. (1993): Über die Rückgewinnung von Sinnlichkeit. Universität Bremen, artec-Arbeitspapier 19

Bruns, W. /Heimbucher, A. /Müller, D. (1993): Ansätze einer erfahrungsorientierten Gestaltung von Rechnersystemen für die Produktion, Universität Bremen, artec-Arbeitspapier 21

Caudill, M. (1992): Kinder, Gentler, Computing. Natural I/O technologies make your computer work for you, instead of the other way around. In: BYTE, Vol. 17, No. 4. pp. 134-150

Gamma, E. (1992): Objektorientierte Software-Entwicklung am Beispiel von ET++. Design-Muster, Klassenbibliothek, Werkzeuge. Berlin, Heidelberg.

Geary, M. (1990): Gemeinsame Quelltexte für PM, Windows und Machintosh. In: Microsoft System Journal. Heft 4

Gorlen, K.E.(): An Objekt-oriented Class Library for C++ Programs. Software Practice and Experience 17, no.12, pp. 899-922

Jerchel, P. (1992): Virtuality now!, Rechnerinterface für den Nintendo PowerGlove, Teil 2-4. In: c't 10-12/92, S. 250-255, 274-280, 278-282

Kaufmann, A. / Yagel, R. / Bakalash,R. (1990): Direct Interaction with a 3D Volumetric Environment. In: Computer Graphics, Vol. 24, No. 2, pp 33-34

Kraus, M. (1992): Virtuality now!, Rechnerinterface für den Nintendo PowerGlove, Teil 1. In: c't 9/92, S. 158, 162

Meyer, A. (1992): Developing a Portable C++ GUI Class Library. In: Dr. Dobb's Journal. Vol. 17, Nov. 92, Issue 11, pp. 102-109

Meyer, B. (1990): Objektorientierte Softwareentwicklung, München.

Rosenstein, L. / Terry, J. (1991): S.:OOP Architectures, MacApp and THINK Class Library, Part one. In: MacTutor, Vol. 7, No. 1., pp. 14-23

Rosenstein, L. / Terry, J. (1991): S.:OOP Architectures, MacApp and THINK Class Library, Part two. In: MacTutor, Vol. 7, No. 3., pp. 8-16

Rubine, D. (1991): Specifying Gestures by Example. In: Computer Graphics, Vol. 25, No. 4, pp 339-348

Schmucker, K.J.. (1989): Object-Oriented Programming for the Macintosh, Hayden, Hasbrouck Heights, New Jersey

Schultz, E. E. (1992): Hawaii International Conference on System Sciences. In: SIGCHI Bulletin Vol. 24, No. 3, pp. 24

Stroustrup, B.. (1991): The C++ Programming Language. Addison-Wesley Publishing Co., Reading, MA

Symantec Corporation (1989): THINK C, Object-Oriented Programming Manual, Symantec Corporation, Cupertino, CA

Valdés, R. (1992): Sizing up Application Frameworks and Class Libraries. In : Dr. Dobb's Journal. Vol. 17, Oct. 92, Issue 10, pp. 18-35

Weinand, A./ Gramma, E./ Marty,R. (1988): ET++ - An objekt- oriented application framework in C++, OOPSLA '88 Proceedings, ACM, pp. 46-57

Wisskirchen, P. (1990): Object-Oriented Graphics. From GKS and PHIGS to Object-Oriented Systems. Symbolic Computation, Springer, Berlin

Weston, D. (1990): Elements of C++ Macintosh Programming. Addison-Wesley Publishing Co., Reading, MA

8. Anhang

Liste aller Quelldateien:

Die in diesem Anhang dokumentierten Angaben beziehen sich auf die Version 0.8. des dargestellten Prototyps und sollen zum besseren Verständnis der Quellprogramme betragen.

Nr	File	c-File	h-File	Klasse	Funktion	por-tabel
1	C3DShape	x	x	(x)	3D-Objekte	
2	CApp	x	x	(x)	Applikation/Core-class	
3	CController	x	x	(x)	Benutzungschnittstelle	
4	CCube	x	x	x	3D-Objekt/Würfel	
5	CDialog	x	x	x	Dialoge	
6	CGDialog	x	x	x	Gesten-Dialog	
7	CGesture	x	x	x	Gesten -Manager	
8	CGKeys	x	x	x	Tastastur/Glove -Manager	
9	CGlove	x	x	x	Glove	
10	CKeys	x	x	x	Tastatur-Manager	
11	CMyWorld	x	x	x	User-Modell/Welt	
12	CObj	x	x	(x)	Root Klasse	
13	CSensors	x	x	(x)	IO - Devices	
14	CSimul	x	x	x	User-Applikation	
15	CSupervisor	x	x	(x)	Kommando-Manager	
16	CView	x	x	(x)	Graf. Repräsentation	
17	CWindow	x	x	x	Fenster	
18	CWorld	x	x	(x)	Modellwelt	
19	CubeDefs		x		Konturdefiniton/Würfel	
20	Deglitch	x	x		Fehleranalyse	
21	Global	x	x		Globale Definitionen	
22	Handdefs		x		Konturdefiniton / Glove	
23	Headers		x		Standard-Headers	
24	Main	x			Hauptprogramm	
25	PG_IO	x	x		IO-Kopplung/Glove	
26	SysEnv		x		Maschinenspez. Headers	
27	UserCmds		x		Definition von Kommandos	
28	Util	x	x		Diverse Utilites	
29	Vector	x	x		Vectorberechnung	
30	AuxIO	x	x		Serielle Schnittstelle	nein
31	DosStuff	x	x		Dos-spezif. Routinen	nein
32	MacStuff	x	x		Mac-spezif. Routinen	nein

Anmerkungen:
Virtuelle bzw. abstrakte Klassen in Klammern

Konventionen für Bezeichner:

In den Quelldateien wurden für Variablen, Klassenbezeichner und Konstanten folgende Konventionen benutzt:

	Präfix	Beispiel	Bemerkung
Klassennamen	C	CView	
Instanzenvariable	its	itsHeight	meist private
Klassenvariable	c	cCount	
Globale Variable	g	gSystem	
Konstante	k	kVersion	
User-Datentyp	t	tWorld	
Variable	-	theName, aName	
Macintosh Variable	mac	macMenubar	
#defines	-	MAX_VALUE	
Befehls-Variablen	cmd	cmdQuit	Parameter für DoCommand
Gesten-Variablen	gest	gestHelp	

Alle Methoden, Funktionen und Klassenbezeichner beginnen mit Großbuchstaben, Variablen und Konstanten immer mit Kleinbuchstaben.