GRAVITATION

Peter Speckmayer DSK540//gravit

1 Prinzip

Das hier beschriebene Programm simuliert die Bewegungen von Planeten mit Hilfe der Newton'schen Formel zur Berechnung der Gravitation:

$$F = \frac{m_1 * m_2}{r^2} * G$$

Wobei m1 und m2 die Massen der momentan berechneten Planeten und r der Abstand zwischen diesen ist. G ist die Gravitationskonstante und beträgt 6.67*E-11 [m^3/(kg*s²)]. Mittels der Massen kann dann die Kraft in eine Beschleunigung umgerechnet werden.

2 Das Universum im Programm

Von jedem Planeten kann eine x, y und z-Position im Universum, ein Geschwindigkeitsanteil in jede dieser drei Richtungen eine Masse, ein Radius (der im Moment aber keinen Einfluß auf die Berechnungen hat) und eine Farbe (mit der der Planet am Bildschirm dargestellt wird) angegeben werden. Weiters kann die Genauigkeit der Berechnungen über die Zeit geregelt werden, hier erfolgt die Angabe in Vielfachen von Sekunden.

In den „#defines" die sich ganz am Anfang des Programmes befinden ist es möglich, die Genauigkeit der Daten der Planeten zu variieren indem man die Zeile #define TYP long double verändert. Statt long double können Datentypen wie float eingesetzt werden, um die Genauigkeit der Berechnung zu verringern, was wiederum die Rechengeschwindigkeit erhöht.

3 Der Aufbau des Programmes

3.1 Allgemeines

Das Programm ist objektorientiert programmiert und könnte theoretisch circa 65000 Planeten (oder Sonnen) verwalten, was allerdings bei der heutigen Rechnergeschwindigkeit nicht sinnvoll ist (günstig ist eine Anzahl bis 10 Planeten). Außerdem wird die Anzahl der Planeten durch den zur Verfügung stehenden Speicherplatz eingeschränkt.

3.2 Die Liste der Planeten

Um mit mehreren Planeten arbeiten zu können, benötigt man eine Liste in der die Planeten verzeichnet sind und mit der man die Planeten verwalten kann. Von Compilererzeugern wie BORLAND oder MICROSOFT werden verschiedene Listenklassen angeboten. Da ich nur ungenügende Beschreibungen dieser Klassen finden konnte, habe ich mir eine eigene programmiert. Die Klasse ist in der Datei liste_x.h zu finden. Sie ist eine template-Klasse, was den Vorteil hat, daß die Klasse TPlanet (in der die Daten für den Planten verzeichnet sind) nachträglich abgeändert werden kann, ohne daß dies auf die Funktion der Listenklasse Einfluß hätte.

Die Liste besteht im Prinzip aus aneinandergereihten Elementen. Die Liste hat einen Kopf und einen Schwanz, dazwischen befinden sich die restlichen Elemente in denen die Planeten verzeichnet sind. Für jeden Planeten gibt es ein eigenes Element. In jedem Element sind die Adressen des nächsten und des vorigen Elementes gespeichert und somit verkettet.

```
[Kopf-Element1-Element2-Element3-...-ElementX-Schwanz]
```

Die verschiedenen Elemente werden durch die Klasse TListe erzeugt und verwaltet. Mit den in TListe enthaltenen Funktionen können neue Elemente erzeugt, bestehende Elemente gelöscht, das aktuelle Element gelesen, das aktuelle Element verändert und ähnliche Tätigkeiten ausgeführt werden.

Weiters ist in der Datei liste_x.h eine Iterator-Klasse enthalten, mit der auf die verschiedenen Elemente einer Klasse zugegriffen werden kann. Es können beliebig viele Iteratoren initialisiert werden. Um einen Iterator zu erstellen, muß man ihn zuerst erstellen TIterator *I1;, wobei der Name des Iterators (hier I1) natürlich beliebig ist. Danach hat man ihn zu initialisieren, was mit Hilfe einer Funktion der Klasse TListe zu geschehen hat

```
Iterator_initialisieren (TIterator *Iterator)
```

3.2.1 Erstellen und verwalten einer neuen Liste

Will man neue Liste erstellen, muß man zuerst eine Klasse erstellen die dann in die Liste eingebettet werden soll. In unserem Beispiel heißt die neue Klasse [TPlanet]. Eine neue Liste kann man jetzt mit der Zeile

```
TListe<TPlanet> Planeten;
```

erstellen. Um Planeten hinzuzufügen, sind in der Listenklasse Operatoren definiert, die diese Aktion erleichtern. So kann man einen neuen Planeten hinzufügen, indem man die Zeile

```
++Planeten;
```

hinzufügt. Mit dem Operator & kann man auf den neu erstellten Planeten direkt zugreifen. Hat man in der Klasse TPlanet beispielsweise eine Variable "Masse" definiert, kann man folgendermaßen auf diese zugreifen:

```
&Planeten->Masse = 330000000000;
```

Hat man bereits mehrere Planeten erstellt, so kann man mit diesem Operator jedoch nur auf den zuletzt erstellten zugreifen, jetzt kommt die Iterator-Klasse (TIterator) ins Spiel. Zuerst hat man einen Iterator zu erzeugen, was man mit folgender Zeile zustande bringt:

```
TIterator I1;
```

Es können, wie schon früher erwähnt beliebig viele Iteratoren definiert werden. Um den Iterator jetzt auch verwenden zu können, muß man ihn initialisieren. Das geschieht folgendermaßen:

```
Planeten.Iterator_initialisieren (&I1);
```

Wird ein neuer Planet erstellt oder ein alter weggelöscht, so muß der Iterator neu initialisiert werden. In der Iterator-Klasse (TIterator) habe ich einige Funktionen programmiert, mit denen der Iterator vernünftig verwendet werden kann.

So kann er mit der Funktion I1=I1.Anfang(); auf das erste Element gesetzt werden, wobei alle Iteratoren einer Liste untereinander kompatibel sind, das heißt, man kann auch folgende Zeile schreiben:

```
I2=I1.Pos();
```

, wobei der Iterator I2 auf dieselbe Position gesetzt wird, an der der Iterator I1 momentan steht. Eine weitere Funktion ist I1.Ende(); mit der das letzte Element angezeigt wird. Die Funktionen Anfang(), Pos(), Ende() liefern immer eine int-Zahl.

Der Iterator kann auch mit = auf eine beliebige Position gesetzt werden I1=...;. Man kann den Iterator inkrementieren und dekrementieren mit den gewohnten Operatoren ++ und --: ++I2; oder --I1;. Um jetzt auf ein Element der Liste mit Hilfe des Iterators zuzugreifen, muß man folgendermaßen vorgehen. Man erhält die Adresse des gewünschten Planeten mit Hilfe des Operators >>.

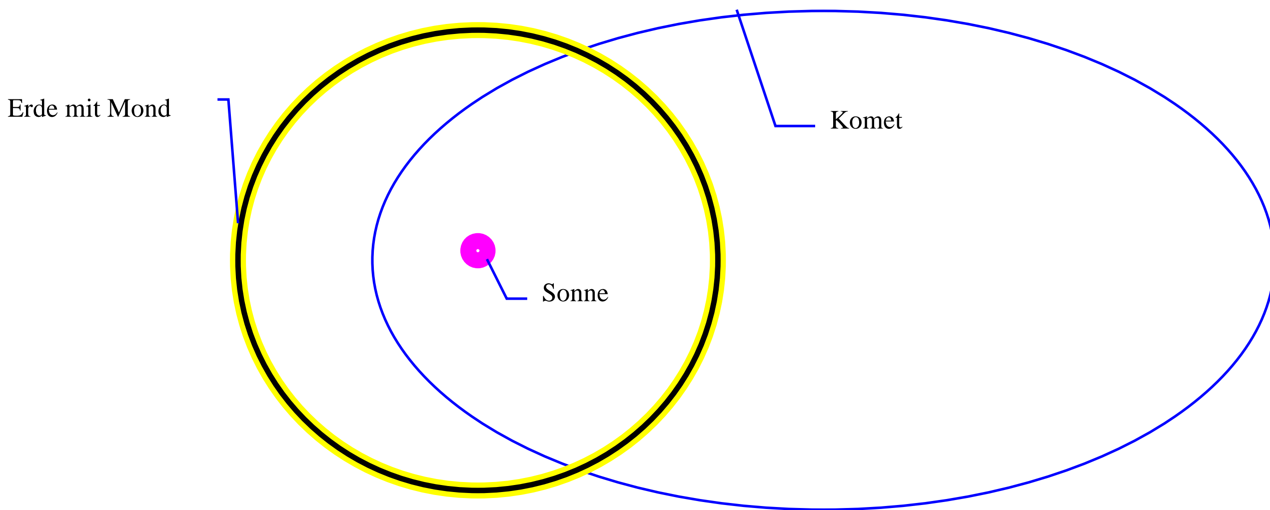
```
Adresse=(Planeten>>I1);
```

(Wobei Adresse: TPlaneten *Adresse; ist.) Um jetzt auf die Variablen und die Funktionen des einzelnen Planeten zuzugreifen, muß man diese nur noch anreihen, beispielsweise:

```
Wert = (Planeten>>I1)->Masse;
```

genauso kann man die Werte ändern: (Planeten>>I1)->Masse=Wert;. Selbstverständlich kann nur auf die im public-Teil der Klasse definierten Daten und Funktionen zugegriffen werden.

Die Listenklasse hat noch einige Funktionen die eher unwichtig sind, auf sie will ich hier nicht näher eingehen, da sie auch im Programm Gravitation nicht verwendet werden.



3.3 Das Gravitationsprogramm

3.3.1 Allgemeines

Wie schon ganz zu Beginn erwähnt, wird die Kraft zwischen jeweils zwei Planeten berechnet und mit Hilfe der Massen in eine Beschleunigung umgewandelt. Die aktuelle Geschwindigkeit des Planeten wird um die Beschleunigung verändert. Indem man jetzt die Geschwindigkeit mit der eingestellten Zeit multipliziert, kommt man auf die Entfernung die der Planet während dieser Zeit zurücklegt.

3.3.2 Beliebig viele Planeten

Für die Programmierung des Programmes bestanden mehrere Probleme. Eines der Probleme war, daß nicht nur zwei Planeten vorhanden waren, sondern, daß das Programm auch für beliebig viele andere Planeten klaglos funktionieren sollte.

Dazu wird im Programm für jeden Planeten berechnet, welchen Einfluß die anderen Planeten haben. Nachdem diese Berechnungen für jeden Planeten ausgeführt worden sind, werden die daraus resultierenden Beschleunigungen einberechnet. Danach wird die Planetenbewegung aller Planeten für die eingestellte Zeit berechnet.

Damit die Änderungen eines Planenten sich nicht schon auf die anderen Planeten auswirken bevor auch die anderen Planeten verändert worden sind, habe ich eine zweite Liste eingeführt in der die Änderungen vermerkt werden. Sind dann die Änderungen für alle Planeten durchgerechnet, dann werden die notierten Änderungen in die Daten des Planeten eingerechnet.

3.3.3 Drei Koordinatenachsen

Die ganz zu Beginn des Artikels erwähnte Formel von Newton muß für die Aufgabenstellung, wie sie sich hier gestellt hat, adaptiert werden, da die Kraft `F`, die man als Ergebnis der Gleichung bekommt, in die Richtung des anderen Planeten zeigt. Damit kann man natürlich nicht viel anfangen. Mein Programm spaltet die Kraft `F` in drei Teilkräfte entlang jeder der Koordinatenachsen auf.

Mit Hilfe der Masse wird dann eine Beschleunigung in jede einzelne der Koordinatenachsen ermittelt. Daraus läßt sich dann die Änderung der Geschwindigkeit in jede Richtung errechnen. Mit der Zeit verknüpft, kann man dann die Positionsänderung für jede der drei Koordinaten berechnen.

3.3.4 Die Zeit

Die Genauigkeit der Planetenbahnen hängt sehr stark von der Zeit ab, die zwischen den Berechnungen liegt. In meinem Programm kann man die Zeit in Vielfachen von Sekunden angeben (zB. 0.0001 oder 3600). Außerdem kann eine statische oder eine dynamische Zeitbasis gewählt werden.

Für jedes spezielle Problem muß man sich überlegen, welche Zeitbasis am günstigsten ist. Die Bewegung der Erde um die Sonne könnte beispielsweise mit einer Zeitbasis von 86400 berechnet werden (86400 s sind ein Tag). Für die Bewegung des Mondes um die Erde wird dieser Wert schon fast zu gering, da der Mond ja doch nur eine Umlaufzeit von ca. 28 Tagen um die Erde hat, hier wäre eine Zeitbasis von 1 h (=3600s) vielleicht günstiger.

```
Uni.ZeitQuantDauer_setzen (STATISCH, 3600, 3600);
```

Außerdem kann man die Zeit noch dynamisch regeln lassen, das funktioniert aber noch nicht so ganz wie ich es gerne hätte, der Ansatz ist aber schon vorhanden.

Bei der dynamischen Zeitregelung kann man eine ober und eine Untergrenze festlegen zwischen denen sich die Zeitdauer verändern kann.

```
Uni.ZeitQuantDauer_setzen (DYNAMISCH, 0.05, 8600);
```

3.3.5 Neue Planeten, andere Daten

Will man einen neuen Planeten erzeugen sind in der Prozedur folgende Zeilen hinzuzufügen:

```
Uni.Planet_erzeugen();
(&Uni.Planeten)->benennen ("Venus");   // Planetenname
(&Uni.Planeten)->Masse = 500E15;       // Planetenmasse in kg
(&Uni.Planeten)->Koordinaten->x = 190E8;
(&Uni.Planeten)->Koordinaten->y = 780E9;
(&Uni.Planeten)->Koordinaten->z = 9E11;
(&Uni.Planeten)->Geschwindigkeit->x = 996;
(&Uni.Planeten)->Geschwindigkeit->y = 921.558E2;
(&Uni.Planeten)->Geschwindigkeit->z = 11E2;
(&Uni.Planeten)->Radius = 6E6;         // Radius in m
                                       // (hat keinen Einfluß)
(&Uni.Planeten)->Farbe = 5;            // Planetenfarbe
```

Momentan sind im Programm folgende Himmelskörper programmiert:

Die Sonne an der Position 0 im Universum und mit der Geschwindigkeit 0. Die Erde und der Mond mit jeweils Position, Geschwindigkeit und Masse, die so genau sind wie ich sie aus diversen Büchern herauslesen habe können, wobei sich die Daten in den verschiedenen Büchern ein bißchen unterscheiden. Ein Komet der in der Sonne verglühen oder die Erde zerstören würde, wenn diese Möglichkeit im Programm vorgesehen wäre. Kollisionen habe ich nicht programmiert (um die Erde vorerst vom Untergang zu bewahren).

3.3.6 Ergänzungen

Ich habe es nicht geschafft, eine funktionierende Graphikausgabe für dieses Programm mit den Borland-C++-Befehlen der Header-Datei `graphics.h` zu programmieren. Aus diesem Grund habe ich ein paar einfache Routinen selbst programmiert (z.B. Darstellung eines Pixels am Bildschirm, Füllen des Bildschirmes). Das Programm funktioniert nur unter DOS.

Programm

```
LISTE_X.H
/*  Liste!
TListe ist eine Template-Klasse mit deren Hilfe man eine einfache Liste
erstellen und verwalten kann.
(programmiert von Peter Speckmayer (1995))
*/


#ifndef LISTE_X__H
#define LISTE_X__H
```

```
template <class TInhalt_des_Elementes>
class TElement            // Klasse für ein Element
{
  public:

    TInhalt_des_Elementes *Inhalt_des_Elementes;
    // Zeiger auf den Inhalt des Elementes
    // (class-Typ wird über die Template-
    // Definition übergeben
    TElement *naechstes;      // Zeiger auf das nächste Element
    TElement *voriges;        // Zeiger auf das vorige Element

    TElement ()     {}        // Konstruktor
    ~TElement ()              // Destruktor
    {                         // löschen des reservierten Speicherplatzes
      if ((this!=voriges)&&(this!=naechstes))
      {
         delete Inhalt_des_Elementes;
      }
    }
};
```

```
class TIterator
// Ein Iterator dient dazu,
// ein bestimmtes Element in der Liste anzusprechen
// Es können beliebig viele Iteratoren gleichzeitig initialisiert sein
// und auf ein jeweils beliebiges Element zeigen.
{
  private:

    int Nummer;     // Die "Nummer" des Elementes,
                    // auf das der Iterator zeigt.
    int Obergrenze; // "Nummer" des letzten Elementes
    int Untergrenze;          // "Nummer" des ersten Elementes
    // die Ober und die Untergrenzen können auch verändert werden, um
    // einen Iterator nach oben oder nach unten zu begrenzen.

  protected:
  public:

    TIterator () { Nummer=Untergrenze=Obergrenze=0; }
       // Konstruktor vor dem Initialisieren.
    TIterator (int Un, int Ob, int Nu)
       // wird beim Initialisieren aufgerufen
    {
      Nummer= Nu; Obergrenze= Ob; Untergrenze= Un;
    }
    ~TIterator () {}          // Destruktor

    int Pos ()    { return Nummer; }     // Auslesen der Nummer
    int Ende ()   { return Obergrenze; } // Auslesen des Endes
    int Anfang () { return Untergrenze; } // Auslesen des Anfanges
    void Pos_setzen (int Nu) { Nummer=Nu; } // verändern der Nummer
    void Anfang_setzen (int Ob) { Obergrenze=Ob; }
      //verändern der Obergrenze
    void Ende_setzen (int Un) { Untergrenze=Un; }
      //verändern der Untergrenze

    void Werte (int Un, int Ob, int Nu)
    {
      Untergrenze=Un; Obergrenze=Ob; Nummer=Nu;
    }  // ver„ndern aller Werte geschlossen

    // Definition von operatoren
    void operator=(int Wert); // Nummer auf einen bestimmten Wert setzen
    void operator++(); // Erhöhen der Nummer um 1
    void operator--(); // Erniedrigen der Nummer um 1
};
```

```
void TIterator::
operator=(int Wert)
{
  Nummer= Wert;
}
```

```
void TIterator::
operator++()
{
  if (Nummer<=Obergrenze)
  {
    Nummer+=1;
  }
}
```

```
void TIterator::
operator--()
{
  if (Nummer>Untergrenze)
  {
    Nummer-=1;
  }
}


// Klasse für die eigentliche Liste
/* Diese Klasse ist als template-Klasse ausgeführt um nicht an eine
bestimmte Datenstruktur gebunden zu sein. Der Inhalt eines Elementes
kann jede noch so komplexe Klasse sein.
*/
```

```
template <class T>
class TListe                   // Hauptklasse
{
  private:
  protected:
  public:

    TElement<T> *Kopf;        // Zeiger auf den Kopf der Liste
    TElement<T> *Schwanz;     // Zeiger auf das Ende der Liste
    TElement<T> *Aktuelles;   // Zeiger auf das Aktuelle Element
    int Anzahl;     // Anzahl der Elemente in der Liste
                    // (ohne Kopf und Schwanz)

    TListe ();      // Konstruktor
    ~TListe ();     // Destruktor

    void dazu (T * Zeiger_auf_den_neuen_Inhalt);
                    //hinzufügen eines Elementes
    void weg ();    // entfernen eines Elementes
    T* lesen ();    // lesen der Adresse des Inhaltes eines Elementes
    void erstes_Element ();
      // setzen des Aktuelles-Zeigers auf das erste Element
    void letztes_Element ();
      // setzen des Aktuelles-Zeigers auf das letzte Element
    void naechstes_Element ();
      // setzen des Aktuelles-Zeiger auf das nächste Element
    void voriges_Element ();  // wie vorher nur auf das vorige Element

// Definition von Operatoren
// zur verkürzten Schreibweise einiger Funktionen
    void operator++() { dazu (new T ()); } // Hinzufügen eines Elementes
    // hinter dem aktuellen Element (Aktuelles)
    void operator--() { weg (); }
    // Entfernen eines Elementes (Aktuelles)
    void operator-(TIterator *Iterator);
    // Entfernen des Elementes auf das der Iterator zeigt.

    void Iterator_initialisieren (TIterator *Iterator); // initialisieren
    // eines Iterators. (es können beliebig viele initialisiert werden.

    // Zwei sehr praktische operator-Definitionen:
    T* operator >> (TIterator Iterator);
    // Element-Adresse lesen, auf die der angegebene Iterator zeigt.
    // z.B.   ListenKlasse>>(Iterator1)->Inhalt1 = 5; (Zuweisung)
    // oder   x = ListenKlasse>>(Iterator2)->Inhalt2;
    // (Auslesen eines Wertes)
    T* operator & ();  // Zeiger auf den Elementinhalt von "Aktuelles"
    // beim neuerstellen eines Elementes kann man nach dem erstellen mit
    // diesem operator direkt auf das erstellte Element zugreifen.
    // z.B.   ++ListenKlasse; (erstellen eines neuen Elementes
    //        &ListenKlasse->Inhalt1 = 7;
    //        &ListenKlasse->InhaltText = "Hallo";
    //        (zugreifen auf das aktuelle Element)
};
```

```
template <class T>
void TListe<T>::
TListe () // Konstruktor
{
  Kopf= new TElement<T> ();   //Speicherbereich für den Kopf reservieren
  Schwanz= new TElement<T> ();
   //Speicherbereich für den Schwanz reservierern

  Kopf->naechstes = Schwanz;
   // Kopf und Schwanz adressenmäßig verknüpfen
  Kopf->voriges = Kopf;
  Schwanz->naechstes=Schwanz;
  Schwanz->voriges = Kopf;    // Aktuelles(Element) ist der Kopf
  Aktuelles= Kopf;

  Anzahl = 0;       // es sind  vorerst keine Elemente vorhanden
}
```

```
template <class T>
void TListe<T>::
~TListe () // Destruktor
{
  Aktuelles = Schwanz->voriges;
   // Freigeben des besetzten Speicherbereiches
  while (Aktuelles!=Kopf)
  {
    weg ();
  }
  delete Kopf;
  delete Schwanz;
  Anzahl = 0;
}
```

```
template <class T>
void TListe<T>::
dazu (T * Zeiger_auf_den_neuen_Inhalt)
{
  TElement<T> *Neues;
  Neues= new TElement<T> ();

  Neues->Inhalt_des_Elementes = Zeiger_auf_den_neuen_Inhalt;
  Neues->naechstes = Aktuelles->naechstes;
  Neues->voriges = Aktuelles;
  Aktuelles->naechstes = Neues;
  Neues->naechstes->voriges = Neues;
  Aktuelles = Neues;

  Anzahl++;
}

template <class T>
void TListe<T>::
weg ()
{
  TElement<T> *Merken;
  if ((Aktuelles!=Kopf)&&(Aktuelles!=Schwanz))
  {
    Merken = Aktuelles;
    Aktuelles->voriges->naechstes = Aktuelles->naechstes;
    Aktuelles->naechstes->voriges = Aktuelles->voriges;
    Aktuelles = Merken->voriges;
    delete Merken;
    Anzahl--;
  }
}

template <class T>
void TListe<T>::
operator-(TIterator *Iterator)
{
  int n;
  TElement<T> *Merken; // Um das aktuelle Element zu merken
  TElement<T> *IteratorElement;  // Adressenzählspeicher
  Merken = Aktuelles;  // Aktuelles wird gemerkt

  IteratorElement = Kopf->naechstes;   // Adressenz. an die erste Pos.
  for (n=0; n<Iterator->Pos(); n++)    // Iterator durchzählen
  {
    IteratorElement = IteratorElement->naechstes; // nächstes Element
  }

  Aktuelles = IteratorElement;  // Aktuelles ist das im Iterator
  weg ();  // eingestellte Element, weglöschen

  if (Merken != IteratorElement) // Wenn Merken nicht das gelöschte
  {                     // Element ist:
    Aktuelles = Merken; // Aktuelles wieder richtigsetzen
  }
  else  // sonst
  {                                     // Aktuelles Element ist das dem
    Aktuelles = IteratorElement->naechstes; // gelöschten Element
  }                                         // nachfolgende Element.
                                        // Neuinitialisieren des
  Iterator_initialisieren (Iterator);   // Iterators.
}

template <class T>
void TListe<T>::
erstes_Element ()
{
  Aktuelles=Kopf->naechstes;
}

template <class T>
void TListe<T>::
naechstes_Element ()
{
  Aktuelles=Aktuelles->naechstes;
}

template <class T>
void TListe<T>::
voriges_Element ()
{
  Aktuelles=Aktuelles->voriges;
}

template <class T>
void TListe<T>::
letztes_Element ()
{
  Aktuelles=Schwanz->voriges;
}

template <class T>
T* TListe<T>::
lesen ()
{
  return Aktuelles->Inhalt_des_Elementes;
   /// Die Adresse des Element-inhaltes wird übergeben */
}

template <class T>
void TListe<T>::Iterator_initialisieren (TIterator *Iterator)
{
  Iterator->Werte (0, Anzahl, 0);
}

template <class T>
T* TListe<T>::operator >> (TIterator Iterator)
// Element-Adresse lesen
{
  int n;
  TElement<T> *IteratorElement;

  IteratorElement = Kopf->naechstes;
  for (n=Iterator.Anfang(); n<Iterator.Pos(); n++)
  {
    IteratorElement = IteratorElement->naechstes;
  }
  return IteratorElement->Inhalt_des_Elementes;
}

template <class T>
T* TListe<T>::operator & ()
// Zeiger auf den Elementinhalt von "Aktuelles"
{
  return Aktuelles->Inhalt_des_Elementes;
}

#endif
```

```
grav.cpp
/* Gravitationsberechnung objktorientiert programmiert */
/* von Peter Speckmayer */

#include <conio.h>

#include <math.h>
#include <stdio.h>
#include <string.h>
#include "liste_x.h"

#define GRAVITATIONSKONSTANTE 6.67E-11
#define TYP float   // TYP definiert die Genauigkeit der Berechnung
#define DIM_ANZAHL 3

#define DYNAMISCH 1 // Modi für die Größenänderung der ZeitQuantDauer
#define STATISCH 0  // dynamische Veränderung oder fixer Wert

#define EIN 1
#define AUS 0

#define word unsigned int
#define byte unsigned char far
#define Pixel_x 320
#define Pixel_y 200
#define Adresse_des_Bildschirmspeichers 0xA000
#define Pixel_pro_Bildschirm 64000


/* die folgenden drei Routinen sind kleine Graphikroutinen zur
Initialisierung, zum Setzen eines Punktes und zum Füllen des Bildschirmes
mit einer Farbe. Ich habe sie deshalb in das Programm eingefügt, da
die Graphikausgabe mit den C++-eigenen Routinen ("graphics.h") aus
irgendeinem unerfindlichen Grund bei mir nicht funktioniert hat. Es steht
natürlich jedem frei die Graphikausgabe zu modifizieren. Die
Graphikausgabe findet in der Klasse KDarstellung (wie der Name schon
sagt) statt. */

void
Graphikmodus_setzen (byte m)
{
 asm mov AH,00h
 asm mov AL,m
 asm int 10h
}

void
Pixel_darstellen (word Position, word Farbe)
{
 asm push AX    // Register auf Stack sichern
 asm push DS
 asm push DI

 asm mov AX,Adresse_des_Bildschirmspeichers // Adr. des B.sp. auf AX
 asm mov DS,AX       // AX --> DS (Datensegment auf B.sp.-Beginn
 asm mov AX,Position // übergebene Pixelposition auf AX
 asm mov DI,AX       // ==> Position auf DI
 asm mov AX,[DI]     // Ausgew‚hlte Speicherstelle auf AX kopieren
 asm xor AL,AL       // niedrigeres byte von AX auf 0 setzen
 asm add AX,Farbe    // Farbe (00|Farbe) zu AX addieren ==> (byte|Farbe)
 asm mov [DI],AX     // AX auf Speicherstelle zurückschreiben

 asm pop DI          // gesicherte Register von Stack holen
 asm pop DS
 asm pop AX
}

void
Bildschirm_fuellen (word Farbe)
{
 asm push AX       // Register auf den Stack sichern
 asm push BX
```

```
 asm push DS
 asm push DI

 asm mov AX,Adresse_des_Bildschirmspeichers
 asm mov DS,AX     // Datensegment auf B.sp.-Adresse legen

 asm mov AX,Farbe  // Farbe ( byte|byte =word ) auf AX
 asm mov AH,AL     // (AX: byte|Farbe) --> (AX: Farbe|Farbe)
    // (Wortweise Adressierung ist schneller

 asm mov BX,Pixel_pro_Bildschirm
 asm mov DI,BX     // Schleifenzähler für Anz. der Pixel
 lab_B_f:
 asm dec DI  // pro Pixel ein Byte, deswegen
 asm dec DI  // (da Wortweise Adressierung) verminderung
             // des Zählers um zwei Byte
 asm mov [DI],AX  // (AX: Farbe|Farbe)
             // AX-->auf Bildschirm-speicher ausgeben
 asm jnz lab_B_f:  // Wenn DI!=0 ==> Rücksprung

 asm pop DI   // gesicherte Register von Stack holen
 asm pop DS
 asm pop BX
 asm pop AX
}


// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Beginn des Programmes zur Berechnung der Gravitation **************
```

```
class KDimensionen
// in dieser Klasse sind die drei Dimensionen fest-
{   // gehalten. Diese Klasse wird f r die Position und
  public:        // die Geschwindigkeit eines Planeten verwendet.
   // In diesem Programm wird ein karthesisches
    TYP x;         // Koordinatensystem verwendet (eben x, y, z).
    TYP y;
    TYP z;

    KDimensionen (); // Konstruktor für KDimensionen
    ~KDimensionen (); // Destruktor für KDimensionen
};

KDimensionen::KDimensionen ()
// Im Konstruktor werden die drei Dimensionen auf 0 gesetzt.
{
  x=0;
  y=0;
  z=0;
}
```

```
KDimensionen::
~KDimensionen ()
{
}


/*  TPlanet ist die Klasse die einen Planeten definiert, die Daten die
bei jedem dieser Planeten eingestellt werden können sind: die
Koordinaten, die Geschwindigkeit, seine Masse, sein Radius, seine Farbe
(ist eigentlich nur für die Darstellung wichtig, sie könnte in einer
verbesserten Version durch ein Sprite ersetzt werden, daß das Bild des
Planeten zeigt). Außerdem kann man noch einen Namen für den Planeten
bestimmen.
*/
```

```
class TPlanet
{
  public:

    char Name[20]; // Name des Planeten

    KDimensionen *Koordinaten; // Koordinaten des Planeten
    KDimensionen *Geschwindigkeit; // Geschwindigkeit des Planeten
    TYP Masse;  // Masse   // in die jeweilige Richtung des
    TYP Radius; // Radius  // Koordinatensystemes
    int Farbe;  // Farbe in der der Planet in weiterer Folge dargestellt
                // wird.
    TPlanet (); // Konstruktor
    ~TPlanet ();// Destruktor

    void benennen (char Name[]) { strcpy (TPlanet::Name, Name); }
    void Werte (KDimensionen Koord, KDimensionen Geschw,
                TYP M, TYP R);
 // "benennen" und "Werte" sind zwei Prozeduren zum einstellen der
 // gewünschten Werte.
};
```

```
TPlanet::
TPlanet ()
// Konstruktor:  stellt alle Variablen auf einen
{     //      Anfangswert.
  Koordinaten = new KDimensionen (); // Speicherbereich für Koordinaten
  Geschwindigkeit = new KDimensionen (); // und für Geschwindigkeit
  Masse = 0;
  Radius = 0;
  Farbe = 1;
}
```

```
TPlanet::
~TPlanet () // Destruktor
{
  delete Koordinaten; // Die Speicherbereiche für Koordinaten und
  delete Geschwindigkeit; // Geschwindigkeit werden freigegeben.
}
```

```
void TPlanet::
Werte (KDimensionen Koord, KDimensionen Geschw, TYP M, TYP R)
// Einstellen der Werte
{
  Koordinaten->x = Koord.x;
  Koordinaten->y = Koord.y;
  Koordinaten->z = Koord.z;

  Geschwindigkeit->x = Geschw.x;
  Geschwindigkeit->y = Geschw.y;
  Geschwindigkeit->z = Geschw.z;

  Masse = M;
  Radius = R;
}


/* KUniversum ist das Universum in dem sich alle Planeten bewegen. Das
Universum beinhaltet die Planeten und die Funktionen zum arbeiten mit den
Planeten. Außerdem ist in dieser Klasse die Regelung der Zeit zu finden.
Die Zeitregelung kann statisch oder dynamisch stattfinden, wobei bei der
dstatischen Zeitregelung die absolute Dauer eines "Zeitquantes" angegeben
werden kann (in Sekunden), bei der dynamischen Zeitregelung können die
obere und die untere Grenze zwischen denen die Zeitquantdauer verändert
werden kann angegeben werden. (Die dynamische Zeitregelung verhält sich
leider noch nicht so ganz wie ich es gerne hätte.) */
```

```
class KUniversum // Klasse für das Universum
{
private:

    TListe<TPlanet> P_Aenderungen; // Planeten: Liste 2
    /* Beinhaltet alle Änderungen die auf die Planeten von einer bis zur
    nächsten Zeiteinheit einwirken.  */
    TIterator I1, I2; // Iteratoren für die Liste von Planeten
public:
    TYP ZeitGrundWert_unten; // Untere und obere Grenzen für die
    TYP ZeitGrundWert_oben;  // dynamische Zeitregelung.
    TYP ZeitQuantDauer;      // Dauer einer Zeiteinheit
    TYP ZeitFaktor;  // Faktor für die dynamische Zeitregelung
    int ZeitModus;  // DYNAMISCH oder STATISCH
    TYP Planeten_plus_P_Aenderungen ();
    // Veränderung der Planeten von einer auf die nächste Zeiteinheit.
    void P_Aenderungen__Null (); // Nullsetzen der Änderungen
public:
    TListe<TPlanet> Planeten; // Planeten: Liste 1
    // Die Liste in der alle Planeten mit ihren Daten verzeichnet sind
    KUniversum (); // Konstruktor
    ~KUniversum (); // Destruktor
    void Berechnung_eine_Zeiteinheit ();
    void Planet_erzeugen ();
    void Planet_loeschen (char Name[]);
    void Planet_veraendern ();
    void ZeitQuantDauer_setzen (int Modus, TYP Wert, TYP Oben);
    /* Setzen der Dauer einer Zeiteinheit (in Sekunden) */
    TYP Abstand_berechnen (TPlanet *P1, TPlanet *P2);
    /* Abstand zwischen zwei Planeten berechnen */
    TYP V_Aenderungen_berechnen (TYP Abstand, TYP Alpha, TPlanet *P1,
    TPlanet *P2, TPlanet *PA1, TPlanet *PA2);
    /* Berechnung der Geschwindigkeitsänderung eines Planeten */
    void ZeitRegelung (TYP Beschleunigung, TYP Geschwindigkeit);
    /* Dynamische Zeitregelung */
};
```

```
KUniversum::
KUniversum () // Konstruktor
{
  ZeitModus = STATISCH;
  ZeitQuantDauer = 1;  // eine Sekunde
}
```

```
KUniversum::
~KUniversum () // Destruktor
{
}
```

```
void KUniversum::
ZeitQuantDauer_setzen (int Modus, TYP Wert, TYP Oben)
{
  switch (Modus)
  {
    case STATISCH:
      ZeitModus = STATISCH;
      ZeitQuantDauer = Wert;
      ZeitGrundWert_unten = Wert;
      ZeitGrundWert_oben = Oben;
      ZeitFaktor = 1E23;
      break;
    case DYNAMISCH:
      ZeitModus = DYNAMISCH;
      ZeitQuantDauer = Wert;
      ZeitGrundWert_unten = Wert;
      ZeitGrundWert_oben = Oben;
      ZeitFaktor = 1E50;
      break;
  }
}
```

```
void KUniversum::
Berechnung_eine_Zeiteinheit ()
/* Diese Prozedur berechnet die Gravitation zwischen allen Planeten und
die daraus erfolgenden Geschwindigkeits und Positionsänderungen. */
```

```
{
  TYP best_a1, best_a2, best_v;
  TYP Abstand;
  TYP Alpha;

  P_Aenderungen__Null ();
  best_a1= best_a2= 0;
  // Für jeden Planeten müssen alle Berechnungen durchgeführt werden.
  Planeten.Iterator_initialisieren (&I1);
  Planeten.Iterator_initialisieren (&I2);

  I1=I1.Anfang();
  while ( I1.Pos() < I1.Ende() )
  {
    I2=I1.Pos()+1;
    while ( I2.Pos() < I2.Ende() )
    {
      Abstand= Abstand_berechnen ( (Planeten>>I1), (Planeten>>I2) );
      Alpha = GRAVITATIONSKONSTANTE / (Abstand*Abstand);

      best_a2 =
      V_Aenderungen_berechnen (Abstand, Alpha, (Planeten>>I1),
        (Planeten>>I2), (P_Aenderungen>>I1), (P_Aenderungen>>I2) );
      if (best_a2>best_a1) best_a1 = best_a2;
      ++I2;
    }
    ++I1;
  }
  // * ZeitQuantDauer
  best_v = Planeten_plus_P_Aenderungen ();
  ZeitRegelung (best_a1, best_v);
}
```

```
TYP KUniversum::
Abstand_berechnen (TPlanet *P1, TPlanet *P2)
```

```
/* Berechnung des Abstandes zwischen zwei Planeten */
{
  TYP X, Y, Z, Ergebnis;

  X = ((P1->Koordinaten->x) - (P2->Koordinaten->x));
  Y = ((P1->Koordinaten->y) - (P2->Koordinaten->y));
  Z = ((P1->Koordinaten->z) - (P2->Koordinaten->z));

  Ergebnis = sqrt ((TYP)(X*X)+(Y*Y)+(Z*Z) );
  return Ergebnis;
}
```

```
TYP KUniversum::
V_Aenderungen_berechnen (TYP Abstand, TYP Alpha, TPlanet *P1,
TPlanet *P2, TPlanet *PA1, TPlanet *PA2)
```

```
/* Berechnung der Geschwindigkeits„nderungen der Planeten */
{
  TYP a1, a2;
  TYP cos_Winkel;
//  Beschleunigung
  a1 = Alpha * ZeitQuantDauer;
  a2 = (P1->Masse) * a1;
  a1 = (P2->Masse) * a1;
// x
  if (Abstand!=0)
  {
    cos_Winkel = ((P1->Koordinaten->x) - (P2->Koordinaten->x)) / Abstand;
    PA1->Geschwindigkeit->x += a1*cos_Winkel*(-1);
    PA2->Geschwindigkeit->x += a2*cos_Winkel;
    cos_Winkel = ((P1->Koordinaten->y) - (P2->Koordinaten->y)) / Abstand;
    PA1->Geschwindigkeit->y += a1*cos_Winkel*(-1);
    PA2->Geschwindigkeit->y += a2*cos_Winkel;
    cos_Winkel = ((P1->Koordinaten->z) - (P2->Koordinaten->z)) / Abstand;
    PA1->Geschwindigkeit->z += a1*cos_Winkel*(-1);
    PA2->Geschwindigkeit->z += a2*cos_Winkel;
  }
  else
  {
    cos_Winkel = 0;
    PA1->Geschwindigkeit->x += 0; //a1*cos_Winkel*(-1);
    PA2->Geschwindigkeit->x += 0; //a2*cos_Winkel;
    PA1->Geschwindigkeit->y += 0; //a1*cos_Winkel*(-1);
    PA2->Geschwindigkeit->y += 0; //a2*cos_Winkel;
    PA1->Geschwindigkeit->z += 0; //a1*cos_Winkel*(-1);
    PA2->Geschwindigkeit->z += 0; //a2*cos_Winkel;
  }
  return Alpha * ZeitQuantDauer; // R ckgabewert f r die dynamische
    // Zeitregelunng
}
```

```
void KUniversum::
ZeitRegelung (TYP Beschleunigung, TYP Geschwindigkeit)
```

```
/* Zeitregelung */
{
  TYP aneu;
  if (ZeitModus == DYNAMISCH)
  {
      ZeitQuantDauer = ZeitGrundWert_unten;
      aneu=ZeitFaktor*Beschleunigung*Geschwindigkeit;
      if (aneu<1) aneu=1;
      if (Beschleunigung!=0) ZeitQuantDauer +=
    ((ZeitGrundWert_oben-ZeitGrundWert_unten)/sqrt(aneu));
      else ZeitQuantDauer = ZeitGrundWert_oben;
      if (aneu==1) ZeitFaktor*=10;
      else if (aneu>=(ZeitGrundWert_oben-ZeitGrundWert_unten))
ZeitFaktor/=10;
  }
}
```

```
void KUniversum::
Planet_erzeugen () // erzeugen eines Planeten
```

```
{
  ++Planeten;  // Hinzuf gen eines Planeten
  ++P_Aenderungen; // Hinzufügen des Speicherplatzes für die Änderungen
   // eines Planeten von einer auf die nächste
   // Zeiteinheit. (++ ist ein operator der in
   // der Template-Klasse Liste
}
```

```
void KUniversum::
Planet_loeschen (char Name[])
```

```
{
  Planeten.Iterator_initialisieren (&I1);
  // Planetennamen mit Name vergleichen.
  I1=(I1.Anfang ());
  while ( I1.Pos()<=I1.Ende() )
  {
    if ( strcmp ( ((Planeten>>I1)->Name), Name) == 0 )
    {
      Planeten-(&I1);
    }
    ++I1;
  }
}
```

```
void KUniversum::
Planet_veraendern ()
```

```
// noch nicht ausgeführt.
{
}
```

```
TYP KUniversum::
Planeten_plus_P_Aenderungen ()
```

```
/* Einberechnen der Änderungen die den jeweiligen Planeten betreffen */
{
  TYP zwischen;
  TYP v, best_v;
  v= best_v=0;
  Planeten.Iterator_initialisieren (&I1);
  I1=(I1.Anfang ()); // Iterator an den Beginn setzen
  while ( I1.Pos()<I1.Ende() )
  {
    (Planeten>>I1)->Geschwindigkeit->x +=
    (P_Aenderungen>>I1)->Geschwindigkeit->x;
    (Planeten>>I1)->Geschwindigkeit->y +=
    (P_Aenderungen>>I1)->Geschwindigkeit->y;
    (Planeten>>I1)->Geschwindigkeit->z +=
    (P_Aenderungen>>I1)->Geschwindigkeit->z;

    ((Planeten>>I1)->Koordinaten->x) +=
    (((Planeten>>I1)->Geschwindigkeit->x) * ZeitQuantDauer);
    ((Planeten>>I1)->Koordinaten->y) +=
    (((Planeten>>I1)->Geschwindigkeit->y) * ZeitQuantDauer);
    ((Planeten>>I1)->Koordinaten->z) +=
    (((Planeten>>I1)->Geschwindigkeit->z) * ZeitQuantDauer);

    if (ZeitModus==DYNAMISCH)
    {
      v = fabsl( (Planeten>>I1)->Geschwindigkeit->x )+
          fabsl( (Planeten>>I1)->Geschwindigkeit->y )+
          fabsl( (Planeten>>I1)->Geschwindigkeit->z );
      if (v>best_v) best_v=v;
    }

    ++I1;
  }
  return best_v; // Rückgabewert für die Zeitregelung
}
```

```
void KUniversum::
P_Aenderungen__Null ()
```

```
/* Nullstellen der Änderungen für die Planeten */
{
  P_Aenderungen.Iterator_initialisieren (&I1);
  I1=(I1.Anfang ()); // Iterator an den Beginn setzen
  while ( I1.Pos()<I1.Ende() )
  {

    (P_Aenderungen>>I1)->Koordinaten->x = 0;
    (P_Aenderungen>>I1)->Koordinaten->y = 0;
    (P_Aenderungen>>I1)->Koordinaten->z = 0;

    (P_Aenderungen>>I1)->Geschwindigkeit->x = 0;
    (P_Aenderungen>>I1)->Geschwindigkeit->y = 0;
    (P_Aenderungen>>I1)->Geschwindigkeit->z = 0;

    ++I1;
  }
}
```

```
class KDarstellung // **** Darstellung
```

```
{
  private:
    TIterator I3;
  public:

    KDarstellung ();
    ~KDarstellung ();

    void Bild (KUniversum *Uni);
};
```

```
KDarstellung::
KDarstellung ()
{
  Graphikmodus_setzen (19);
  Bildschirm_fuellen (0x00);
}

KDarstellung::
~KDarstellung ()
{
  textmode (BW80);
}

void KDarstellung::
Bild (KUniversum *Uni)
{
  int X=100, Y=100;
  Uni->Planeten.Iterator_initialisieren (&I3);
//    Bildschirm_fuellen (0x00);
  while (I3.Pos()<I3.Ende())
  {
    X=(int)(((Uni->Planeten>>I3)->Koordinaten->x)/2E9)+160;
    Y=(int)(((Uni->Planeten>>I3)->Koordinaten->y)/2E9)+100;
    if ((X>0)&&(X<320)&&(Y>0)&&(Y<200))
    {
      Pixel_darstellen (X+ (Y*320), (Uni->Planeten>>I3)->Farbe);
    }
    ++I3;
  }
}

void main ()
{
  int n;
  KUniversum Uni;
  KDarstellung Dar;
// Vorgang beim erstellen eines neuen Planeten:
//   Uni.Planet_erzeugen ();
//   (&Uni.Planeten)->Koordinaten->x =  10; .....

  #define ERDM 5.977E24

  Uni.Planet_erzeugen();
  (&Uni.Planeten)->benennen ("Sonne");
  (&Uni.Planeten)->Masse = 333000*ERDM;
  (&Uni.Planeten)->Koordinaten->y=0;
  (&Uni.Planeten)->Koordinaten->x=0;
  (&Uni.Planeten)->Geschwindigkeit->y=0;
  (&Uni.Planeten)->Geschwindigkeit->x=0;
  (&Uni.Planeten)->Farbe = 7;

  Uni.Planet_erzeugen();
  (&Uni.Planeten)->benennen ("Erde");
  (&Uni.Planeten)->Masse = 1*ERDM;
  (&Uni.Planeten)->Koordinaten->y=149.6E9;
  (&Uni.Planeten)->Koordinaten->x=0;
  (&Uni.Planeten)->Geschwindigkeit->y=0;
  (&Uni.Planeten)->Geschwindigkeit->x=29.8E3;
  (&Uni.Planeten)->Farbe = 2;

  Uni.Planet_erzeugen();
  (&Uni.Planeten)->benennen ("Mond");
  (&Uni.Planeten)->Masse = ERDM/81;
  (&Uni.Planeten)->Koordinaten->y=(149.6E9);
  (&Uni.Planeten)->Koordinaten->x=-374E6;
  (&Uni.Planeten)->Geschwindigkeit->y=996;
  (&Uni.Planeten)->Geschwindigkeit->x=29.8E3;
  (&Uni.Planeten)->Farbe = 6;

  Uni.Planet_erzeugen();
  (&Uni.Planeten)->benennen ("Komet");
  (&Uni.Planeten)->Masse = 18E3;
  (&Uni.Planeten)->Koordinaten->y=(179.6E9);
  (&Uni.Planeten)->Koordinaten->x=-100E8;
  (&Uni.Planeten)->Geschwindigkeit->y=-425E2;
  (&Uni.Planeten)->Geschwindigkeit->x=-70.8E2;
  (&Uni.Planeten)->Farbe = 6;

  Uni.ZeitQuantDauer_setzen (DYNAMISCH, 3600, 5*86400); //86400);

  while (!kbhit())
  {
    Uni.Berechnung_eine_Zeiteinheit ();
    Dar.Bild (&Uni);
  }

}
/* Klassen sind durch ein vorangesetztes "K" gekennzeichnet
   (z.B. KDarstellung, ... )
   Klassen die in weiterer Folge in template-Klassen verwendet werden
   sind durch ein vorangestelltes T gekennzeichnet (z.B. TPlanet) */
```

□

NETNEWS EFFIZIENT NUTZEN

Philipp Krone

Wer sich ein wenig länger und intensiver mit dem Internet beschäftigt, stellt oft binnen kürzester Zeit fest, daß das WWW zwar schön bunt ist, in vielen Fällen aber keine brauchbaren oder tiefergehenden Informationen hergibt. Die aktiveren Benutzer (vor allem also jene, die nicht nur im 'Cyberspace surfen' wollen, wie es die Medien stets anpreisen, sondern vom Netz profitieren und Informationen gewinnen wollen) entdecken dann Mail (bzw Mailing-Lists) und News für sich.

An dieser Sache ist nun ein kleiner Haken: Internet-Netnews zu lesen ist über Internet-Zugänge schrecklich ineffizient. Entweder man liest online, da kostet der Spaß eine Menge Online-Zeit (was bei den heimischen Ortsgebühren nicht wirklich erfreulich ist) oder man benützt Programme, mit denen man nach Subjekts selektieren, und die gewünschten Postings erst dann auf den heimischen Rechner holen kann, wo man später die Möglichkeit hat, sie in aller Ruhe zu lesen. Das ist schon mal nicht schlecht, aber immer noch zu langsam, da die Selektion nach Subjects einerseits Zeit und Mühe kostet (und man oft nicht anhand eines Subjects entscheiden kann, ob das Posting lesenswert ist), und andererseits der Transfer ohne jegliche Kompression abläuft.

Die Lösung für den professionellen Netnews-Nutzer heißt also UUCP, da hier die Daten komprimiert und automatisiert übertragen werden, ohne jeglichen Zeitaufwand für den Nutzer, allein: sehr wenige Provider bieten UUCP an, die wenigen aber verlangen oft gesalzene Preise (zB PING *): bis 15 Newsgroups: öS 200/Monat, bis 50 Newsgroups: 300, darüber 400 Schilling pro Monat); zusätzlich fallen für den Nutzer dann meist noch die Gebühren für einen interaktiven Account, der PPP-Sessions ermöglicht, an. Als rühmliche Ausnahme sei hier at.net erwähnt: bei diesem Provider ist UUCP bereits im (auch zeitlich unlimitierten) Account inkludiert.

Preiswerter kann das der aktive Benutzer aber auch über das FidoNet haben. Effizient (weil komprimiert (nicht nur die News, sondern auch Mail, was vor allem für Bezieher von mailing Lists interessant sein dürfte) und damit kostengünstig und zeitsparend. Keine teure Monatsgrundgebühr für einen Provider, keine langen Online-Zeiten. Und das ohne den Verzicht auf irgendetwas. Dazu interssante Fido-Gruppen, und alles in einem Programm.

Das Gateway des Information Technology Club - ITC stellt bereits jetzt sämtliche Newsgroups der at.* - Hierachie und viele andere kostenlos zur Verfügung, jedes 'Advanced'-Mitglied hat darüberhinaus die Möglichkeit, aus einer Liste von über 18.000 Gruppen beliebige weitere zu bestellen. -

*) Stand zum Zeitpunkt der Drucklegung lt. Ping-Hotline.

HOTLINE	*„Mercedes Hotline, was kann ich für Sie tun?“*
KUNDE	*„Guten Tag, ich habe mir soeben mein erstes Auto gekauft und ich habe mich für ein Auto von Ihnen entschieden, weil es Tempomat, Servolenkung, hydraulisch verstärkte Bremsen und elektrische Zentralverriegelung hat.“*
HOTLINE	*„Danke, daß Sie sich für unser Produkt entschieden haben. Was kann ich für Sie tun?“*
KUNDE	*„Wie funktioniert das Auto?“*
HOTLINE	*„Können Sie autofahren?“*
KUNDE	*„Ob ich was kann?“*
HOTLINE	*„Können Sie autofahren?“*
KUNDE	*„Ich bin kein Techniker. Ich will mich nur mit meinem Auto fortbewegen!“*