“Dragon Graphics”
Forth, OpenGL und 3D-Turtle-Graphics

Bernd Paysan

22. April 1999

Zusammenfassung

Eine 3D-Turtle-Graphics auf der Basis von OpenGL wird vorgestellt. Die Turtle bewegt sich im Raum, und zieht eine Spur, die das Skelett der 3D-Objekte definiert. Die Oberfläche wird ausgehend von der Turtle mittels verschiedener Koordinatensysteme (besonders beliebt: Zylinderkoordinaten) gesetzt. Normalenvektor und Texturkoordinaten werden algorithmisch generiert. Anhand zweier Beispiele, eines Baums und des Swap-Drachens, der dieser Technik auch den Spitznamen gibt, wird die Vorgehensweise demonstriert.

1 Einleitung

Auf der letzten Forth-Tagung habe ich eine direkte OpenGL-Anbindung in Forth vorgestellt. OpenGL ist eine 3D-Grafik-Library, die einem viel Arbeit abnimmt. Allerdings ist OpenGL relativ low-level, und bietet „nur“ Koordinatentransformationen sowie das Zeichen von Strips, also Aneinanderreihungen von Dreiecken oder Rechtecken an. Zudem benötigt OpenGL Normalenvektoren und Texturkoordinaten, die man aber automatisch berechnen kann.

Mein Vorhaben war daher, OpenGL in eine einfacher zu benutzende Library zu kapseln, eine Art 3D-Turtle-Grafik. Um den Jahreswechsel gab es eine Diskussion in comp.lang.forth über eine solche 3D-Turtle-Grafik. Dave Taliaferro stellte eine in pForth geschriebene 3D-Turtle-Grafik vor. Marcel Hendrix implementierte kurz darauf etwas vergleichbares in iForth.

Beide Turtles können sich durch den Raum bewegen und hinterlassen dabei eine Spur aus OpenGL-Objekten, etwa Zylindern oder Kugeln. Man kann damit also keine komplexen Körper erzeugen.

Das hier vorgestellte System setzt auf dem Turtle-Prinzip auf, erlaubt es aber, Körper zu beschreiben. Da es diese nicht als Komposition aus starren Einzelteilen aufbaut, ist eine echte Skelettanimation möglich, etwas, was auch bei Hollywood-Tools noch mit viel Aufwand verbunden ist. Nicht zufällig haben die abendfüllenden Streifen Insekten, also Außenskelette, als Darsteller. Animationen mit Innenskeletten beschränken sich auf kurze Sequenzen. 3000 Punkte (der Drache) kann man auch nicht einfach von Hand eingeben.

2 Das Prinzip

Eine normale 2D-Turtle-Grafik kann vorwärts und rückwärts fahren, sowie sich nach rechts und links drehen. Dabei hinterläßt sie Spuren, also Striche. Das Prinzip läßt sich auch auf Flächen erweitern, indem man die von der Turtle gezeichneten Polygone auffüllt.

Im Raum ist die Turtle in ihrem richtigen Element (unter Wasser). Statt schwerfällig herumzukriechen, kann sie auch nach oben und unten schwimmen, sowie um ihre Achse rollen. Man muß sich nun überlegen, wie die „Spur“ aussehen soll, und wie man von Strichen auf Flächen und noch wichtiger Körper kommt.

Statt einfach vorgefertigte Objekte fallenzulassen, erlaubt es diese 3D-Turtle-Grafik, Schnitte durch den Körper zu beschreiben. Diese Schnittebenen werden dann miteinander verbunden, um einen Körper zu formen. Um etwa einen Zylinder darzustellen, verbindet man zwei Kreise miteinander. Kreise werden durch Vielecke angenähert.

Die 3D-Turtle-Grafik stellt für diese Schnitte keine 2D-Turtle-Grafik zur Verfügung (obwohl das ja irgendwie naheliegend wäre), sondern verschiedene Koordinatensysteme, etwa Zylinder-Koordinaten. Man kann natürlich auch die 3D-Turtle verwenden, Umrisse abzufahren. Der Ur-

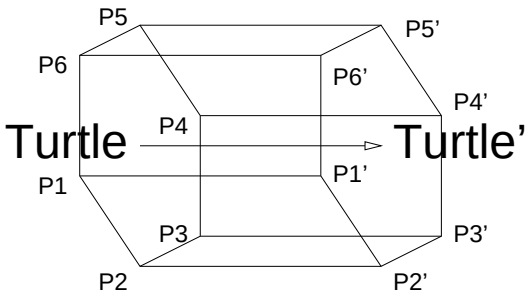

Abbildung 1: 3D-Turtle-Prinzip

sprung ist durch die Turtle bestimmt, die Ausrichtung des Koordinatensystem entspricht der Blickrichtung der Turtle.

3 Ein einfaches Beispiel

Als einfaches Beispiel soll uns ein Baum dienen. Ein Baum besteht aus einem Stamm und Zweigen, die wir hier durch mit Sechsecken angenäherten Zylindern darstellen. Als Blatt soll eine einfache Kugel-Näherung dienen. Unser Baum hat ein paar Parameter: die Verzweigungstiefe, und die Anzahl der Äste. Der oben dargestellte Baum hat auch noch eine Wahrscheinlichkeit, mit der Äste ausfallen, die wollen wir hier aber nicht implementieren.

Fangen wir also mit dem Stumpf an. Zunächst brauchen wir eine untere Begrenzungsfläche, hier erst einmal ein Sechseck. Wir lassen die Turtle, wo sie gerade steht, und öffnen einen Pfad mit sechs Punkten pro Runde.

```
: baum ( m n -- )
  .brown .color  6 open-path
```

Die Sechsecke haben einen Winkel von $\pi/3$ pro Schritt, den können wir uns schon mal vormerken. Er bestimmt die Schrittweite für die Funktionen, die keinen Winkel als Parameter haben.

```
  pi 3 fm/ set-dphi
```

Nun fangen wir mit sechs Punkten in der Mitte an. Wir müssen zunächst die sechs Punkte hinzufügen (der Pfad ist am Anfang leer), und dann in der nächsten Runde nochmal setzen, um die Normalenvektoren richtig zu setzen (aller Anfang ist schwer — da die Normalenvektoren sich auf die letzte Runde beziehen, gibt es in der ersten Runde noch gar keine).

```
  6 0 DO  add  LOOP next-round
  6 0 DO  set  LOOP next-round
```

Um sie herum werden in der nächsten Runde

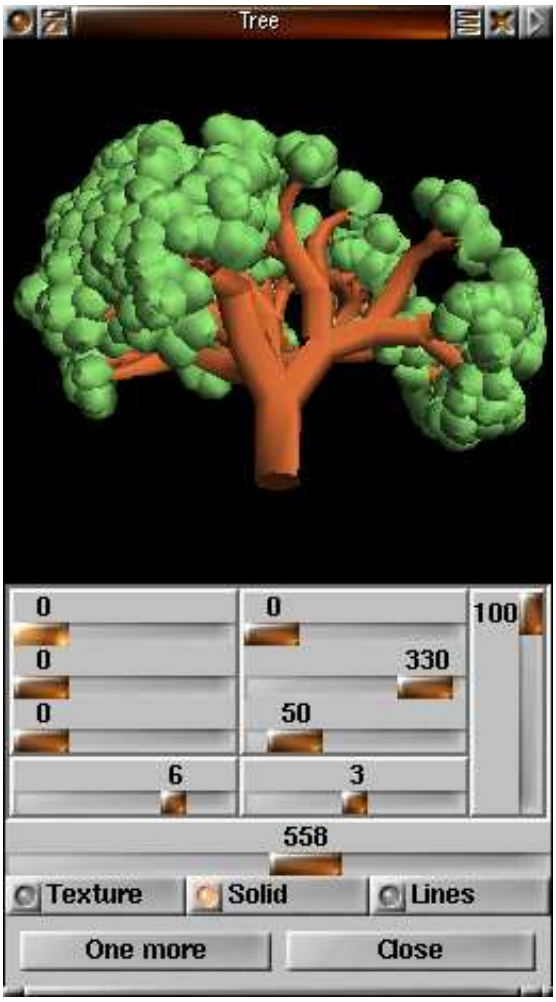

Abbildung 2: Baum

die Dreiecke gezeichnet, die das Boden-Sechseck ergeben. Die Größe der Dreiecke ist hier aus der Verzweigungstiefe errechnet, indem die mit 0.03 multipliziert wird. Da OpenGL selbst mit Fließkommazahlen arbeitet, verwendet auch die Turtle-Grafik solche Zahlen.

```
  6 0 DO dup !.03 fm* set-r LOOP next-round
```

Nun kommt ein kleiner Trick, um eine scharfe Kante zu setzen - die 3D-Turtle-Grafik berechnet nämlich die Normalenvektoren an einem Punkt aus der Summe der Kreuzprodukte der Vektoren nach links/hinten und nach rechts/vorne. Ein weiterer Schnitt an derselben Stelle bewirkt, daß nur eine Richtung für den Normalenvektor berücksichtigt wird.

```
  6 0 DO  dup !.03 fm* set-r  LOOP
```

Nun können wir zum eigentlichen rekursiven Teil kommen, den Zweigen:

```
  zweige ;
```

```
: zweige ( m n -- )  recursive
```

Um eine doppelte Rekursion zu vermeiden, verwende ich eine Schleife für die Endrekursion.

```
  BEGIN  dup  WHILE
```

Auch hier müssen wir erstmal eine neue Runde anfangen. Damit der Baum nicht so plattgepreßt in der Ebene steht, drehen wir ihn pro Verzweigung um 54 Grad.

```
        next-round  pi !.3 f* roll-left
```

Als nächstes müssen wir entsprechend der Verzweigungstiefe vorwärts gehen, und einen neuen Ring zeichnen.

```
        dup !.1 fm* forward
        6 0 DO dup !.03 fm* set-r LOOP
```

Für die weiteren Verzweigungen brauchen wir eine Schleife — bis auf die letzte Verzweigung, die wird ja von der Endrekursion abgearbeitet.

```
        over 1 ?DO
```

Jeder Ast wird durch Rotieren um die Blickachse gedreht — `I'` ist hier das Ende der Schleife. Der Befehl `>turtle` sichert den aktuellen Status der Turtle auf einem Turtle-Stack, `turtle>` nimmt ihn wieder herunter. Ich verwende eine lokale Variable, da die Turtle etwas Returnstackplatz braucht, und damit `I` und `I'` nicht verfügbar sind. Den Fließkommastack darf man auch nur für Zwischenberechnungen verwenden, da die C-Library von einem leeren Stack ausgeht.

Nach der Drehung müssen wir nach rechts (um 18 Grad hier), und danach die Turtle wieder zurückdrehen - damit die Punkte der jeweiligen Schnitte zusammenpassen. Die geänderte Blickrichtung der Turtle bleibt durch diese Operation erhalten, nur ihre Ausrichtung im Raum wird zurückgesetzt.

```
            2pi I I' fm*/ { f: di |
            >turtle
               di roll-left pi 5 fm/ right
               di roll-right
               2dup 1- zweige
            turtle> }
```

So, nun noch die Schleife fertigmachen

```
            LOOP
```

und für die Endrekursion nach rechts kippen (diesmal ist die Drehung 0 Grad).

```
        pi 5 fm/ right
  1- REPEAT
```

Am Schluß noch den Pfad zumachen, und ein Blatt zeichnen.

```
  close-path leaf 2drop ;
```

Das Blatt selbst ist eine einfache angenäherte Kugel:

```
  : leaf ( -- )
    .green .color
    6 open-path  6 0 DO  add  LOOP
    next-round !.1 forward
        6 0 DO  !.2 set-r  LOOP
    next-round !.2 forward
        6 0 DO  !.2 set-r  LOOP
    next-round !.1 forward
        6 0 DO  !.1 set-r  LOOP
    next-round
        6 0 DO  !0  set-r  LOOP
    close-path .brown .color ;
```

Das sind noch nicht die ganzen Sourcen, wir brauchen noch etwas Overhead, um die Ansicht auf den Baum zu verändern. Die ganzen Sourcen finden sich in der Datei `tree.str` (3D-Grafik) und `tree.m` (Benutzeroberfläche).

4 Ein komplexeres Beispiel: Der Drache

Da der Drache sehr komplex ist, beschreibe ich hier nur die wesentlichen Punkte. In typischer Forth-Tradition wird der Drache vom Schwanz her aufgezäumt.

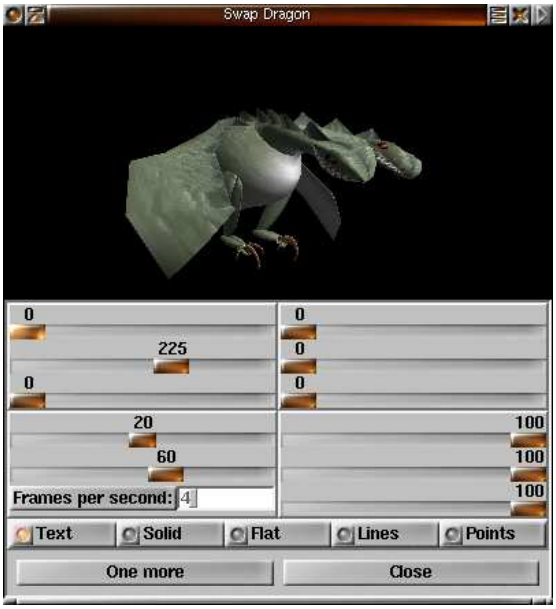

Abbildung 3: Swap-Drache

4.1 Schwanz

Der Drache besteht aus einzelnen Segmenten, die im wesentlichen einen Kreis mit einem Zacken darstellen:

```
: dragon-segment ( ri ro n -- )
```

```
  { f: ri f: ro | next-round
    ro set-r  1 DO  ri set-r  LOOP
    ro !-0.0001 set-rp !0 phi df! } ;
```

Damit der Schwanz so schön wackelt, und auch um die anderen Bewegungen zu synchronisieren, gibt es einen Timer, der in einen Winkel $[0, 2\pi[$ umgesetzt wird.

```
Variable tail-time
: time' ( -- 0..2pi )
  tail-time @ &24 &60 &30 * * um* drop
  0 d>f !$2'-8 pi f* f* ;
```

Das eigentliche Schwanzwackeln wird dann aus der Segmentnummer und der Zeit berechnet — das Ergebnis ist die Verschiebung nach links bzw. rechts.

```
: tail-wag ( n -- f )
  >r pi r@ 1 + fm* !.2 f* time' f+
  fsin r> 2+ dup * 1+ fm/ !30 f* ;
```

Der Ursprung des Drachens liegt im Bauch, nicht an der Schwanzspitze. Gezeichnet wird der Drache aber von der Schwanzspitze her — also muß zunächst eine Kompensation berechnet werden, sonst wackelt der Schwanz mit dem Drachen.[1]

```
: tail-compensate ( n -- f )  !0
  0 DO  I 2+ tail-wag f+ !1.1 f/  LOOP
  !1.1 !20 f** f* fnegate ;
```

Der eigentliche Schwanz ist damit recht einfach: erstmal zur Schwanzspitze zurück, und einen Punkt als Anfangspolygon setzen. Dann Schritt für Schritt den Schwanz wackeln lassen, ein Stück vorwärts gehen, und ein Drachensegment zeichnen. Jedes zweite Drachensegment hat einen Zacken nach oben, und die Skalierung macht den Schwanz auch immer dicker. Der Radius wird zusätzlich vergrößert. Diese Skalierung muß natürlich zuerst in die andere Richtung vorgenommen werden. Als Texture-Mapping-Funktion wird z, ϕ verwendet, also Bewegung der Turtle für die eine Texturkoordinate, und der Winkel gegen die Senkrechte für die andere.

```
: dragon-tail ( ri r+ h n -- ri h )
  zphi-texture
  { f: ri f: r+ f: h n |
  !1.05 !-20 f**
  !1.1  !-20 f** !1 scale-xyz
  h -&15 fm* &20 tail-compensate
  h -&25 fm* forward-xyz
  n 1+ 0 DO  add  LOOP
  20 0 DO !0  i 2+ tail-wag  h forward-xyz
      pi &90 fm/ up
      ri fdup I 1 and 0= IF  r+ f+ THEN
      n dragon-segment
      !1.05 !1.1 !1 scale-xyz
      !.025 ri f+ to ri
  LOOP  ri r+ h } ;
```

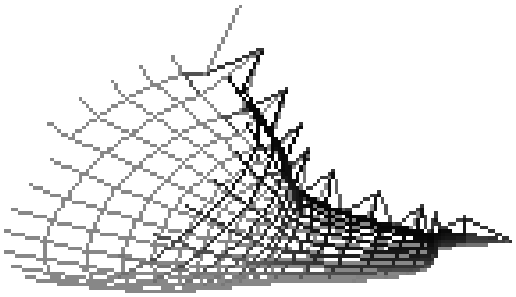

Abbildung 4: Schwanz

4.2 Körper

Der Körper des Drachens besteht aus genau denselben Segmenten wie der Schwanz, nur statt weiter zu wachsen, muß der Körper sich wieder schließen.

```
: dragon-wamp ( ri r+ h ri+ n -- ri' )
  { f: ri f: r+ f: h f: ri+ n |
  8 0 DO  h forward
      ri fdup I 1 and 0= IF  r+ f+ THEN
      n dragon-segment
      ri+ ri f+ to ri !-0.02 ri+ f+ to ri+
  LOOP  ri ri+ !.02 f+ f- } ;
```

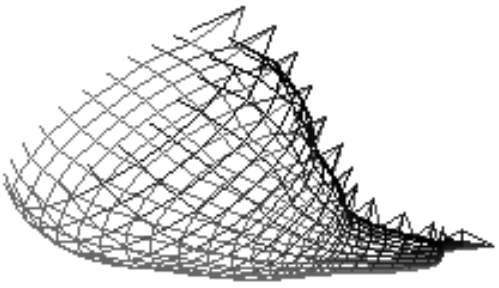

Abbildung 5: Körper

4.3 Hals

Auch der Hals besteht aus diesen Segmenten; allerdings gibt es hier zwei verschiedene Wachstumsfunktionen, eine für die Schulter (schnelle Größenabnahme), und eine für den eigentlichen Hals (langsame Abnahme). Die Schulter biegt

[1] Sowas soll in der Politik schon mal vorkommen.

sich nach links, der Hals wieder nach rechts. Entsprechend wird die Funktion `dragon-neck-part` zweimal aufgerufen.

```
: dragon-neck-part
  ( ri r+ h factor angle n m -- ri' )
  swap { f: ri f: r+ f: h f: factor f: an-
gle n |
  0 ?DO  h forward  angle left
      pi &30 fm/
      time' fsin !.01 f* f+ down
      factor ri f* to ri
      ri fdup I 1 and 0= IF  r+ f+ THEN
      n dragon-segment
  LOOP ri } ;
: dragon-neck ( ri r+ h angle n -- )
  { f: r+ f: h f: angle n |
  r+ h !.82 angle
    n 4 dragon-neck-part
  r+ h !.92 angle f2/ fnegate
    n 6 dragon-neck-part
  fdrop close-path } ;
```

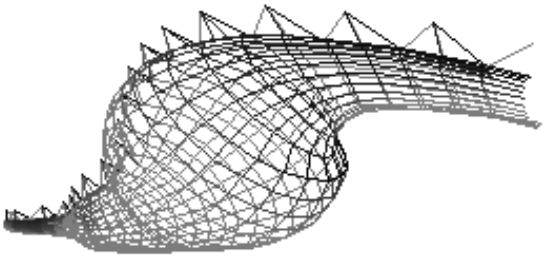

Abbildung 6: Hals

4.4 Kopf

Der Kopf besteht aus einem abgerundeten Rechteck mit einer Einritzung für die Zähne. Die Funktion ist nicht so einfach zu generieren, deshalb verwende ich ein Array für die Koordinaten, allerdings nur für die linke Hälfte des Kopfs; die rechte wird durch Spiegelung an der Y-Achse gewonnen. Die Größenverhältnisse der Schnitte zueinander entsprechen in etwa dem Bauch. Der Kopf hat eine andere Textur, eine mit Augen, Nasenlöchern und Zähnen.

```
Create head-xy
       !0.28 f>fs , !0.0 f>fs ,
       !0.30 f>fs , !0.5 f>fs ,
       !0.25 f>fs , !0.6 f>fs ,
       !0.05 f>fs , !0.6 f>fs ,
       !0.00 f>fs , !0.5 f>fs ,
       !-.05 f>fs , !0.6 f>fs ,
       !-.10 f>fs , !0.6 f>fs ,
       !-.15 f>fs , !0.5 f>fs ,
: dragon-head ( t1 shade -- )  !text
pi 6 fm/ down  !1.2 !.4 !.4 scale-xyz
!-.65 forward
!.5 x-text df!
16 open-path  16 0 DO  add  LOOP
6 0 DO
    I 5 = IF    !.25
          ELSE  I 0= IF !0 ELSE !.35 THEN
          THEN forward
    >matrix
    pi !0.1 f* I 2* 5 - fm* fcos
    fdup !.5 f+ !1 scale-xyz
    next-round
    head-xy 16 cells bounds DO
        I sf@ I cell+ sf@ set-xy
        2 cells +LOOP
    head-xy dup 14 cells + DO
        I sf@ I cell+ sf@
        !1'-6 f+ fnegate set-xy
        -2 cells +LOOP
    matrix>
LOOP
!1 x-text df!
close-path ;
```

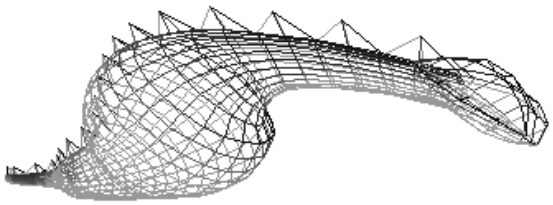

Abbildung 7: Kopf

Der zweite Hals und Kopf werden mit entsprechend negierten Winkeln gezeichnet. Ähnlich dem vorherigen Beispiel wird dazu der Status der Turtle gesichert, und vom selben Status erneut ausgegangen.

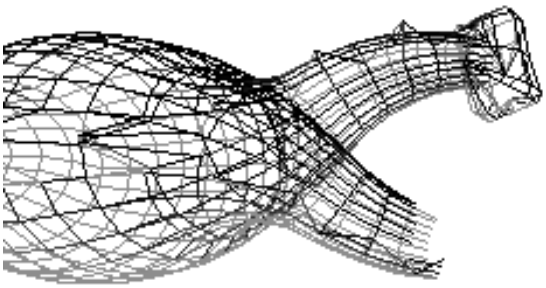

Abbildung 8: Zweiter Hals

4.5 Flügel

Der Flügel hat ein einfaches, flachgestrecktes Sechseck als Schnitt. Dieses Sechseck sorgt für die Krümmung des Flügels, und wird zur Modellierung der „Finger“ verwendet.

```
: wing-step { f: f2 f: f3 |
  next-round
  !0 f2 fnegate                        set-xy
  f3 f2/ f2 fnegate                    set-xy
  f3 f3 !.125 f*                       set-xy
  f3 !.001 f- f3 !.125 f* !.001 f+ set-xy
  f3 f2/ f2                            set-xy
  !0.001 f2 fmin f2                    set-xy }
;
```

Die Falte-Funktion des Flügels sorgt für eine Bewegung von Arm/Unterarm und der Finger abhängig von der Zeit für eine Auf/Abwärtsbewegung des Flügels. f_2 ist ein additioneller Term zum Cosinus, f_1 ein multiplikativer.

```
: wing-fold ( f1 f2 -- )
  time pi 5 fm/ f- fcos f+ f* down ;
```

Die Bewegung und der Aufbau des Flügels sind kompliziert; deshalb erkläre ich nicht alle Einzelheiten. Auch hier wird zunächst ein Pfad geöffnet. Danach werden schrittweise Flügelansatz, Ober- und Unterarm, und zuletzt die drei Finger gezeichnet.

```
: wing ( -- )
  8 open-path !.9 scale
  6 0 DO  add  LOOP
  !.02 !1.2 wing-step !.3 forward      Ansatz
  pi &10 fm/ down  pi &8 fm/ roll-left
  time' fsin !1.3 f* !.2 f+ right
  !.02 !1 wing-step                   Oberarm
  pi 5 fm/ up  pi &10 fm/ right  !1 forward
  pi 5 fm/ down  pi &20 fm/ left
  time' fcos !-.25 f* !.5 f- roll-left
  time' fcos pi 6 fm/ f* down
  !.02 !1 wing-step                 Unterarm
  time' !1 f- fcos !1 f+ pi 8 fm/ f* right
  pi -3 fm/ !-1.0 wing-fold
  pi &10 fm/ left !1 forward
  pi 4 fm/ !-1.5 wing-fold
  !.02 !2 wing-step
  2 0 DO  !.025 forward
          pi &12 fm/ !1.2 wing-fold
          pi &10 fm/ right !.05 forward
          !.02 !2 wing-step            Finger
  LOOP
  !0 !2 wing-step                     Abschluß
  close-path ;
```

Der eigentliche Flügel wird für rechts und links grundsätzlich gleich gezeichnet. Die Symmetrie wird durch eine Spiegelung an der Y-Achse erreicht. Hier muß noch ein Wort zu OpenGL gesagt werden: Nur die Vorderseiten der Dreiecke werden tatsächlich gezeichnet. Durch so eine Spiegelung werden aber aus allen Vorderseiten „Rückseiten“, weil sich die Umlaufrichtung ändert. Also muß man das OpenGL mitteilen, und das macht `flip-clock`.

```
: right-wing ( h -- )
  pi/4 roll-right pi/2 right
  !2 f*  forward pi !.3 f* roll-left
  zp-texture !.13 y-text df! wing ;
: left-wing ( h -- ) !1 !-1 !1 scale-xyz
  flip-clock right-wing flip-clock ;
```

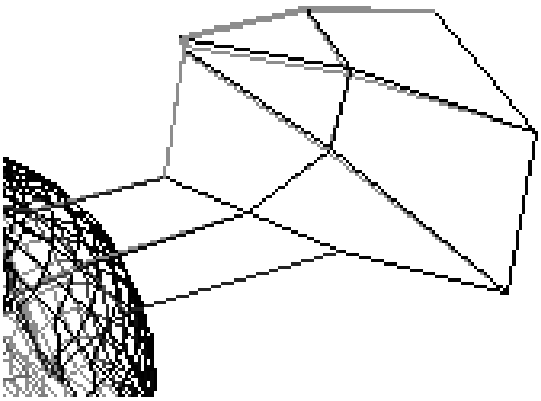

Abbildung 9: Flügel

4.6 Der ganze Drache

Die Beine lasse ich hier mal weg, sie sind nicht so interessant, da sie aus statischen Teilen bestehen (im Wesentlichen langgestreckte Ellipsoide und bogenförmige Klauen). Kommen wir zum Hauptprogramm:

Zunächst wackelt der Drache bei jedem Flügelschlag etwas auf und ab. Dann muß für die Drachensegmente noch der Winkel gesetzt werden.

```
: dragon-body
  ( t0 s t3 s t1 s t3 s t2 s n -- ) >r
  time' fsin !.1 f* !0 !0 forward-xyz
  pi f2* r@ fm/ set-dphi
  r@ 1+ open-path
```

Zuerst wird wie gesagt der Schwanz gezeichnet.

```
  !.1 !.3 !.2 r@ dragon-tail
```

Die Rückgabeparameter des Schwanz werden auch für den Bauch weiterverwendet.

```
  r> { f: ri f: r+ f: h n |
  ri r+ h !.06 n dragon-wamp fdrop
```

Hals und Kopf werden jeweils für rechts und links aufsetzend von derselben Stelle gezeichnet; jeweils mit negierten Winkelparameter

```
>turtle
   ri r+ h  !10 grad>rad n dragon-neck
   2dup dragon-head  2swap !text
turtle>  >matrix
   ri r+ h !-10 grad>rad n dragon-neck
   dragon-head  2drop
matrix>
```

Danach muß die Textur geändert werden, und die beiden Flügel werden gezeichnet.

```
2dup !text
h !2 f* forward
>turtle  h right-wing   turtle>
>turtle  h left-wing   turtle>
```

Analog werden auch die Füße gezeichnet.

```
h !-6 f* forward
>turtle  right-leg   turtle>
>turtle  left-leg    turtle>
2drop 2drop } ;
```

5 Ausblick

Was kann man damit machen, was fehlt noch? Als seriöse Anwendung kann natürlich die Darstellung von dreidimensionalen Daten gelten. „Unseriösere“ Anwendungen wären etwa Computerspiele. Dafür braucht man dann Kollisionserkennung, und wahrscheinlich ein hierarchisches Modell, um Räume und bewegte/bewegbare Objekte einzuordnen. Auch unterschiedliche Levels of Detail abhängig von der Größe des Objekts am Bildschirm müssen jetzt noch mühsam von Hand programmiert werden. Ob es hier für animierte Objekte überhaupt eine andere Möglichkeit gibt, ist mir nicht klar.

Die Behandlung von verschiedenen Texturen ist im Moment noch zu aufwendig; sie müssen auf dem Stack herumgeschleppt werden. Hier muß auch das 3D-Turtle-Objekt selbst noch mehr Tools zur Verfügung stellen.

Und wie immer macht Windows Schwierigkeiten. Obwohl man nicht behaupten kann, daß die MESA-Bibliothek unter Linux fehlerfrei ist, so implementiert sie zumindest alle Features von OpenGL 1.2. Die Windows 95-OpenGL-Library von Microsoft läßt Texturen gleich ganz weg, und funktioniert auch sonst recht wenig zuverlässig. Der Drache wird jedenfalls schwarz auf schwarz dargestellt. Da Silicon Graphics ihre GLX-Sourcen freigegeben haben, sind die verbleibenden Linux-Probleme und die fehlende Hardwareunterstützung (nur 3Dfx wird unterstützt) wahrscheinlich in Kürze ausgeräumt.

Das Ganze kann man downloaden unter
`http://www.jwdt.com/~paysan/bigforth.html`

6 Anhang: Befehle der 3D-Turtle-Grafik

6.1 Navigation

left (f −−) turns the turtle's head left

right (f −−) turns the turtle's head right

up (f −−) turns the turtle's head up

down (f −−) turns the turtle's head down

roll-left (f −−) rolls the turtle's head left

roll-right (f −−) rolls the turtle's head right

x-left (f −−) rotate the turtle left around the x axis

x-right (f −−) rotate the turtle right around the x axis

y-left (f −−) rotate the turtle left around the y axis

y-right (f −−) rotate the turtle right around the y axis

z-left (f −−) rotate the turtle left around the z axis

z-right (f −−) rotate the turtle right around the z axis

forward (f −−) move the turtle in z direction

forward-xyz (fx fy fz −−) move the turtle

degrees (f −−) steps per circle. Common cases: 2π for radians (default), 360 for deg, 64 for asian degrees, or whatever you find suits your application best.

scale (f −−) scales the turtle's step width by the factor f

scale-xyz (fx fy fz −−) scale the turtle's step width in x, y, and z direction

flip-clock (−−) change default coordinate from left hand to right or the other way round. Use that after `scale-xyz` with an odd number of negative scale factors.

6.2 Turtle state

>matrix (−−) push turtle matrix on the matrix stack

matrix> (−−) pop turtle matrix from the matrix stack

matrix@ (−−) copy turtle matrix from the stack

1matrix (−−) initialize turtle state with the identity matrix

matrix* (−−) multiply current transformation matrix with the one on the top of the matrix stack (and pop that one)

clone (−− o) create a clone of the turtle

>turtle (−−) clone the turtle and use it as current object

turtle> (−−) destroy current turtle and pop previos incarnation

6.3 Pathes

open-path (n −−) opens a path with n points in the first round

close-path (−−) closes a path and performs the final rendering action

next-round (−−) closes a round and opens the next one

open-round (n −−) opens a round with n points (obsolete)

close-round (−−) closes a round (by copying the first point as last point) and performs the per-round rendering action (obsolete)

finish-round (−−) performs the per-round rendering action without closing the round first (this is for open objects) (obsolete)

add-xyz (fx fy fz −−) adds the point at the x, y, z-coordinates relative to the turtle. x is up from the turtle, y right, z before. The point is connected to the same point of the previous round as the point before.

set-xyz (fx fy fz −−) sets a point with x, y, z-coordinates. The point is connected to the next point of the previous round as the point before.

drop-point (−−) skips one point, `set-xyz` is equal to `add-xyz drop-point`

set-rpz (fr fphi fz −−) set with cylinder coordinates

set-xy (fx fy −−) set-xyz with $z = 0$

set-rp (fr fphi −−) set with cylinder coordinates, $z = 0$

set-r (fr −−) set with cylinder coordinates, $z = 0$, $\phi = \phi_{cur}$, $\phi_{cur} = \phi_{cur} + \Delta\phi$

set (−−) set at current turtle location

add-rpz (fr fphi fz −−) add with cylinder coordinates

add-xy (fx fy −−) add-xyz with $z = 0$

add-rp (fr fphi −−) add with cylinder coordinates, $z = 0$

add-r (fr −−) add with cylinder coordinates, $z = 0$, $\phi = \phi_{cur}$, $\phi_{cur} = \phi_{cur} + \Delta\phi$

add (−−) add at current turtle location

set-dphi (fdphi −−) sets $\Delta\phi$

6.4 Drawing Modes

points (−−) draw only vertex points

lines (−−) draw a wire frame

triangles (−−) draw solid triangles

textured (−−) draw textured triangles

smooth (−−) variable: set on for smooth normals when rendering textured, set off for non-smooth rendering

xy-texture (−−) texture mapping based on x and y coordinates

zphi-texture (−−) texture mapping based on z and ϕ coordinates

rphi-texture (−−) texture mapping based on r and ϕ coordinates

zp-texture (−−) texture mapping based on z and the point number coordinates

load-texture (addr u −− t) loads a ppm file with the name *addr u* and returns the texture index t

set-light (par1..4 par n −−) Set light source n