"Dragon Graphics"
Forth, OpenGL and 3D-Turtle-Graphics

Bernd Paysan

August 29, 2009

Abstract

A 3D turtle graphics based on OpenGL is presented. The turtle moves in space, and leaves a trail which defines the sceletton of the 3D objects. The surface is created starting at the turle's position using a number of coordinate systems (very popülary: cylinder coordinates). Normal vectors and texture coordinates are computed automatically. Using an example, the swap dragon, the name patron of this technology, the proceeding is demonstrated.

1 Introduction

On the last German Forth-Tagung, I presented direct OpenGL library bindings in Forth. OpenGL is a very powerful 3D graphics library. However, OpenGL is quite low-level, and provides "only" coordinate transformation and drawing of strips, thus strings of triangles or quads. Furthermore, OpenGL needs normal vectors and texture coordinates that could be computed automatically.

My intention was therefore to capsulate OpenGL in an easier to use library, a sort of 3D turtle graphics. Around new year 1999, there was a discussion in comp.lang.forth about such a 3D turtle graphics. Dave Taliaferro introduced a 3D turtle graphics written in pForth. Marcel Hendrix soon afterwards implemented something comparable in iForth.

Both turtles can move through space, and leave a trail composed of OpenGL objects, e.g. cylinders or spheres. You can't compose more complex bodies.

The system introduced here starts with the turtle principle, but it allows to describe bodies. Since it doesn't base on composition of fixed parts, a real sceleton animation is possible, something that even Hollywood tools can only accomplish with a lot of effort. Not accidentally the evening-filling films have insects, thus exosceletons, as actors. Animations with entrosceletons typically restrict to short sequences. 3000 points (the dragon) aren't easy to enter by hand.

2 The Principle

Ordinary 2D turtle graphics can walk forward and backward, and turn right or left. Thereby it leaves trails, thus lines. The principle can be extented to areas by filling the turtle drawn polygons.

In space, the turtle is in its right element (under water). Instead of crawling around clumsy, it can swim up and down and roll around its axis. You just must think about how its "trail" should look like, and how to get from strips and polygons to real bodies.

Instead of dropping pre-factured objects, this 3D turtle graphics allows to describe slices through the body. These slice planes then are connected to form abody. E.g. to create a cylinder, you connect two circles together. Circles are approximated by polygons.

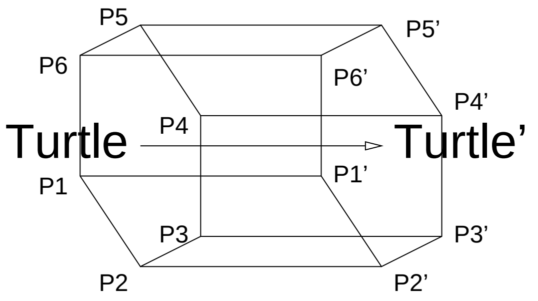

Figure 1: 3D-Turtle-Prinzip

The simple 3D turtle graphics doesn't provide a 2D turtle graphics for these slices (which might be somewhat intuitive), but different coordinate systems like cylinder coordinates. You could use the plain 3D turtle, as well, to draw outlines. The origin is defined by the turtle, the orientation of the coordinate system is where the turtle looks at.

3 A Simple Example

As simple example I'll choose a tree. A tree is composed of a trunk, and branches, which we'll approximate by hexagon-based cylinders. As leaf, I'll use a simple sphere approximation. Our

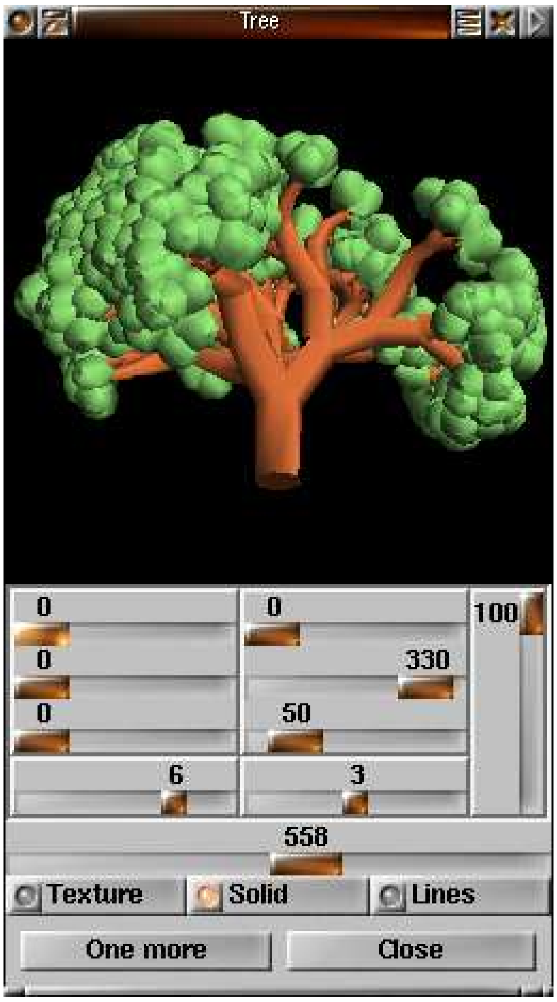

Figure 2: Tree

tree has a few parameters: the branch depth and the number of branches. The tree shown above also has a likelyhood with which the branches fall, we won't implement that here.

Let's start with the root. First, we need a lower surface, a hexagon. We leave the turtle as is, and open a path with six points per round.

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

The hexagons have an angle of $\pi/3$ per step, we can memorize that one now. It defines the step width for the functions that don't take an angle as parameter.

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

Now we start with six points in the middle. We first add the six points (the path is empty at the beginning), and in the next round we set them again, to set the normal vectors correctly (all beginning is difficult — since the normal vectors relate to the previous round, there are none in the first round).

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

Around them in the next round we draw the triangles that form the bottom hexagon. The size of the triangles is computed using the branch depth and multiplied by 0.03. Since OpenGL itself uses floating point numbers, the turtle graphics also works with such numbers.

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

Now I use a small trick to create sharp edges — the 3D turtle graphics computes normal vectors on a point as sum of the cross products of the vectors left/behind and right/forward. Another slice at the same position causes that only one direction is considered for the normal vectors.

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

Now we can procede with the real recursive part, the branches:

```
  branches ;
: branches ( m n -- )  recursive
```

To avoid a double recursion, I use a loop for the end recursion.

```
  BEGIN  dup  WHILE
```

Even here we must start with a new round. To avoid that the tree is flat in a single plain, we roll it every branch by 54 degrees.

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

Next, we got corresponding to the branch depth forward to draw a new ring.

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

For the other branches we need a loop — except for the last branch, that is done by the endrecursion.

```
         over 1 ?DO
```

Each branch rotates around the turtle's eye axis — `I'` here is the end of the loop. The word `>turtle` saves the current state of the turtle on a turtle stack. `turtle>` takes it back again. I use a local variable, since the turtle needs some return stack, and therefore `I` and `I'` aren't accessible. The FP stack can be used only for intermediate computations, since the C library expects an empty stack.

After rotating we must turn right (with an angle of 18 degree here), and then turn the turtle back — so that the points of each slice fit together. The changed eye direction of the turtle remains with this operation, only the alignment in space is reconstructed.

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

Just finish the loop

```
          LOOP
```

and turn right for the end recursion (this time we roll by 0 degrees).

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

Finally, we close the path and draw a leaf.

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

The leaf itself is a simple approximation to a sphere:

```
:  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 ;
```

These aren't all the sources, we need some overhead to change the view to the tree. The whole sources are in the file `tree.str` (3D graphics) and `tree.m` (user interface).

4 A More Complex Example: The Dragon

Since the dragon is really complex, I describe only the most important points. In typical Forth tradition the dragon is briddled by the tail.

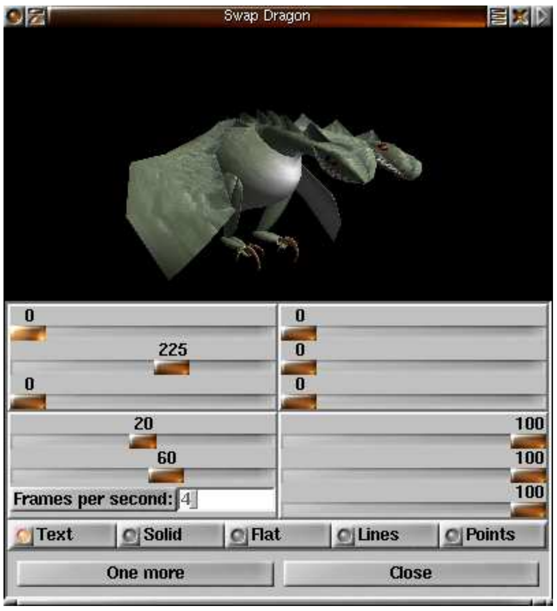

Figure 3: Swap-Drache

4.1 Tail

The dragon is composed of single segments, wich mainly are a circle with a point:

```
:  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!  } ;
```

To wag the tail nicely, and to synchronize all the other movements, there's a timer that is turned into an angle $[0, 2\pi[$.

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

The real tail wagging now computes using segment number and time — the result is a translation left or right.

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

The origin of the dragon is in the womb, not at the tail's point. The dragon however is drawn beginning with the tail's point — thus we first must compute a compensation, otherwise the tail wags with the dragon.[1]

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

The tail then is quite simple: first back to the tail's point, and set a point as initial polygon.

[1]Something like that somethimes happens in politics.

Then, step by step forward, and draw a dragon segment. Each second segment has a point upwards, and scaling makes the tail thicker and thicker. The radius furthermore is enlarged, too. This scaling first has to be undertaken into the other direction. As texture mapping function, I use z, ϕ, thus the movement of the turtle as one texture coordinate and the angle against vertical for the other.

```
:  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 } ;
```

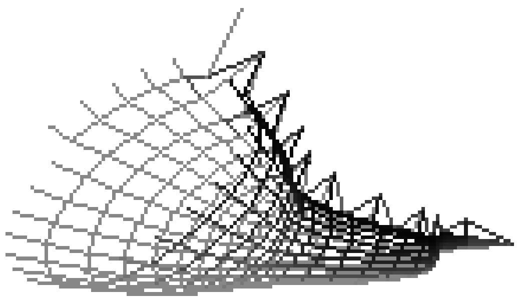
Figure 4: Tail

4.2 Body

The dragon's body is composed out of the same segments as the tail, but instead of growing further, the body must close again.

```
:  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- } ;
```

4.3 Neck

The neck also consists of these segments, however, we have here two different growth functions, one for the shoulder (fast shrink), and one for the real neck (slow shrink). The shoulder turns left, the neck turns right again. Therefore the function `dragon-neck-part` is called two times.

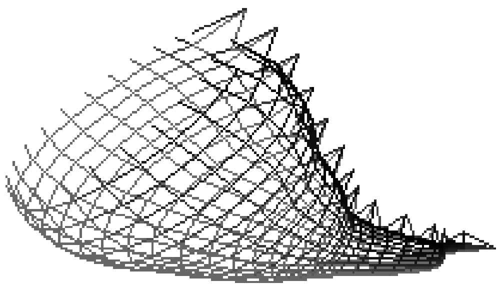
Figure 5: Body

```
:  dragon-neck-part
  ( ri r+ h factor angle n m -- ri' )
  swap { f:  ri f:  r+ f:  h f:  factor f:
angle 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 } ;
```

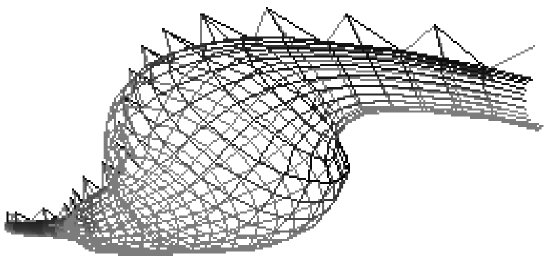
Figure 6: Neck

4.4 Head

The head is composed using a rectangle with rounded edges and a slot for the tooths. This function isn't easy to generate, therefore I use an array for the coordinates, bu tjust for the left half of the head; the right half is obtained by mirroring at the Y axix. The sizes of the slices is about the same as for the body. The head has a different texture, one with eyes, nose, and teeth.

```
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 ;
```

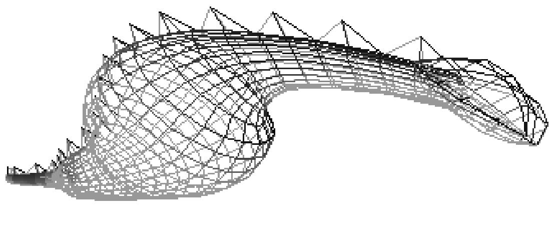
Figure 7: Head

The second neck and head are drawn with corresponding negative angles. Like in the previous example, I save the state of the turtle to start from the same point again.

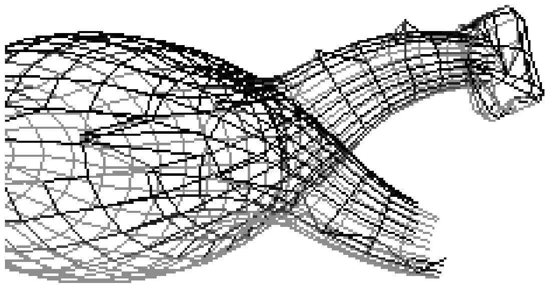
Figure 8: Second Neck

4.5 Wing

The wing has a simple, flat hexagon as slice. This hexagon provides the bending of the wing, and models the "fingers".

```
:  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 }
;
```

The folding function of the wing supplys a movement of arm/hand and finger dependent on time for a up/downward movement of the wing. f_2 is an additional term to the cosine, f_1 a multiplicative.

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

The movements and the composing of the wing are complicated; therefore I don't explain all details. Here first I open a path, too. Then, step by step, shoulder, arm, and finally the fingers are drawn.

```
:  wing ( -- )
  8 open-path !.9 scale
  6 0 DO  add  LOOP
  !.02 !1.2 wing-step !.3 forward   Shoulder
  pi &10 fm/ down  pi &8 fm/ roll-left
  time' fsin !1.3 f* !.2 f+ right
  !.02 !1 wing-step                 Upper arm
  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                 Lower arm
  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             Closing step
  close-path ;
```

The wing is the same left and right. The symmetry is created by mirroring on the Y axis. Here, I must say another word about OpenGL: only the front sides of triangles really are drawn. The backfaces are culled. Such a mirror operation turns all fronts into "backs", since the turn changes. So I must tell OpenGL, and that's what`flip-clock` does.

```
:  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 ;
```

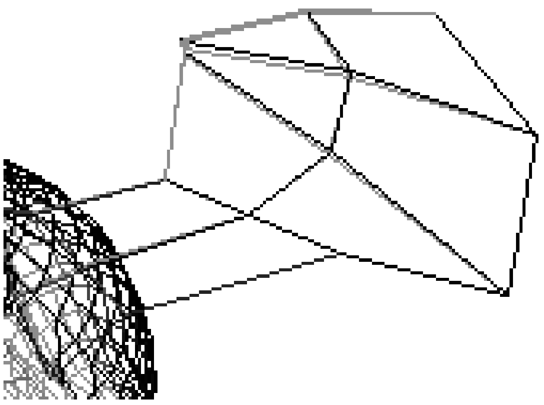

Figure 9: Wing

4.6 The Complete Dragon

I'll leave the legs out here, they aren't that interesting, since they consist of static parts (mostly long ellipsoids and bowed claws). Let's see the main program:

First of all, the dragon wags up and down with each wing fold. Then, for the dragon segments I must set the angle.

```
:  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
```

Then as said above, I draw the tail.

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

The return parameters of the tail are recycled in the body.

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

I draw head and neck left and right starting at the same position, with negated angle parameters for the other side.

```
>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>
```

Then, the texture changes and I draw the two wings.

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

I draw the legs with the same approach.

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

5 Outlook

What can you do with that, and what's missing? A serious application is certainly the visualization of three dimensional data. Less "serious" applications would be computer games. They require collision detection, and quite likely a hierarchical model, to put spaces and moving/moveable objects in. Also different level of detail depending on the size of the object on the screen now must be programmed by hand. If there is another possibility for animated objects I'm not sure.

The usage of different textures is quite complex at the moment; they must be carried on the stack. Here, the 3D turtle object should provide better tools.

And again, Windows makes difficulties. Even though one can't say that the Mesa library under Linux is bug-free, it at least implements all features of OpenGL 1.2. The Windows 95 OpenGL library from SGI/Microsoft omits textures, and doesn't work very reliably. Since SGI opened up their GLX sources, the remaining Linux problems and the missing hardware support (only 3Dfx supported now) will be solved in the near future.

You can download all that under
`http://www.jwdt.com/~paysan/bigforth.html`

6 Appendix: Instructions of the 3D Turtle Graphics

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