1 %!

Any entry into a magical word starts with a code word to make the gates open. In the case of PostScript the magic password is the one above. Thus a PostScript code should always start with **%!** on the first line, and this means literally. No nothing before, not even a space. (And it is also a good practice that a new line should start as everything behind a % is like in tex (conveniently) ignored by the compiler.) The reason is that when a file is sent to the printer via the unix command

```
lpr -Pmathps* file.ps
```

the printer is then told that it is a PostScript file, otherwise it will only treat the code as text and may in some cases churn out some 400 pages of code (this has indeed happened to me).

Also every code should end on the last line with **showpage** otherwise the printer will churn out nothing. The command **showpage** tells the printer that the page has been formatted and is ready to be printed.

However, when viewed via a previewer one may be more lax, the initial code as well as the concluding 'showpage' are not necessary. Among the previewer `gs` is the best, although not quite as good as its old version. It does give error messages invaluable to the programmer (the old one also presented the picture up to the confusing command greatly faciliating diagnosis, while the new one simply dumps the picture as soon as it runs into problem). The most modern previwers like `ggv` are useless. If there is some mistake, however trivial, it only returns 'invalid PostScript code' and quits. This is clearly not intended for PostScript programmers, a rare breed indeed, we are told, of unusually peculiar people.

2 Objects and procedures and stack manipulation

A program basically consists in writing down sequentially objects and procedures. Objects are things like numbers *3 4.0 3.14159* or strings *(please) (do) (not) (read) (this)* or more special things like fonts */ZapfChancery-MediumItalic* , while procedures or functions act on the objects. The objects are arranged sequentially in stacks. When you write down an object it is automatically put on top of the stack. And whenever a function or procedure is being used, it applies itself on the objects of the stack sequentially starting with the most recent (topmost). In stacks, as we will notice, all kinds of objects may be mixed.

The result of a procedure is usually not visible to you, only a few procedures result in things that can be printed out. But of course the purpose of the game is to produce at the end things you can see on screen, or better still view on paper.

The first things we have to do is to familiarize ourselves with some simple procedures that manipulate stacks, providing the basic internal machinery so to speak. To add an object to a stack we only write say *5* then we recall that this number will be the topmost object. So the basic procedures are the following

12 45 67 **pop** $\Longrightarrow$ *12 45*

5 7 43 **dup** $\Longrightarrow$ *5 7 43 43*

2 7 1 8 **exch** $\Longrightarrow$ *2 7 8 1*

Note that those only involve the topmost objects of a stack (of course repeated application of **pop** will kill the entire stack producing nothing). For a more intrusive rearrangement we need the **roll** operator, which affects a cyclical permutation

5 7 43 8 3 1 **roll** $\Longrightarrow$ *5 8 7 43*

The first number *3* tells us how deep into the stack we want to penetrate, while **roll** *3 1* means putting the topmost object on the third place affecting a cyclic permutation moving everything to the right of the cutoff one (1) step rightwards, similarly **roll** *3 2* means moving it

two (2) steps rightwards. Thus e.g

1 2 3 4 5 6 5 3 **roll** $\Longrightarrow$ *1 3 4 5 6 2 3*

Thus *N 0* **roll** will do nothing at all, while *N N* **roll** will work but in the end accomplishing nothing. Also of course *2 1* **roll** is the same thing as **exch** . All the above could be used on any kinds of objects. Then there are arithmetical procedures only meant for numbers with the predictable results.

4 5 **add** $\Longrightarrow$ *9*
4 5 **mul** $\Longrightarrow$ *20*
4 5 **div** $\Longrightarrow$ *0.8*
4 5 **idiv** $\Longrightarrow$ *0*
24 7 **mod** $\Longrightarrow$ *3*

where the last two only works on integers. It is also a good thing to only involve positive integers, computer languages are notoriously bad as handling negative numbers in modular aritmetic. Incidentally **idiv** and **mod** are good converting pairs of numbers to single numbers and back in a way the reader can easily figure out.

Note, and this may have to be emphasized. Procedures only works to the left, i.e. on the 'past' anything in the future is unknown to it. Also procedures are of course not placed on the stack thus the command

roll *3 1* **exch**

would be impossible.

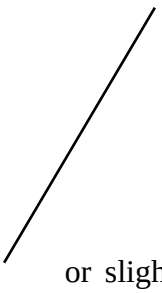
3 Drawing lines and curves

The paper on which we draw comes with a co-ordinate system. The left hand lower corner corresponds to *(0,0)* while the height of the paper is approximately *800* and the width *600* . The precise depends on the setting of the printer and can easily be experimentally checked. (On the screen the limits are *(610,800)* while on the paper we have *(580,830)* with nothing to the left or below of 10 does not print out). It is a good idea to have some safety of margin. The units of measurements may appear somewhat haphazard, in fact 72 corresponds to one inch.

The drawing of a line involves a few procedures. The first being **newpath** which informs that a new line is about to be drawn. A line has a starting point given by regular co-ordinates, and an ending point, likewise given by co-ordinates. The procedure is completed by telling the machine when to stop and draw. Thus the code is as follows

% draws a line from *(40,60)* to *(300,500)*

newpath *40 60* **moveto** *300 500* **lineto stroke**



We could also have done it as follows

newpath *40 60* **moveto** *260 440* **rlineto stroke**

where **rlineto** refers to 'relative'.

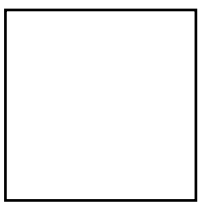
A simple rectangle is drawn by

newpath *100 100* **moveto** *0 50* **rlineto** *50 0* **rlineto** *0 -50* **rlineto** *-50 0* **rlineto stroke**

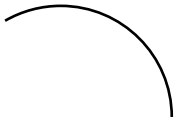
or slightly simpler by

newpath *100 100* **moveto** *0 50* **rlineto** *50 0* **rlineto 0 -50 rlineto closepath stroke**

An arc of a circle is specified by the center of the circle, its radius and the initial angle and the final angle. Thus



newpath *100 100 50 0 120* **arc stroke**



and hence a (full) circle can be coded as

newpath *100 100 50 0 360* **arc stroke**

Those procedures with **arc**

are handy, but strictly speaking not necessary. In fact curves can always been drawn using polygons, which when sufficiently finely meshed, are indistinguishable to the human eye from a smooth curve. PostScript actually comes with a full tool-set of curve-fitting procedures, as well as various dotted lines, which I have never used. The point is that with a few simple procedures you can do almost anything. Drawing a line between two points is the most basic graphic operation you can think of. There is also another, not quite as basic, but not derivable from the basic procedures, namely fillings, which is fundamental for more serious graphing work. In fact when closed curves are drawn one may also fill them, a very useful device in 3-D imaging producing hidden lines. Also if you want to draw very small figures, it is much better to fill them, than to draw their contours, as the thickness of the line will be too dominant. (As we will see you can also vary the thickness of lines, but in extreme cases this might be more of a theoretical exercise than a practical one, as there seems to be limits to the printers ability to draw very thin lines, to say nothing about the screen; yet small fillings the printer seems to be able to do down to the cut-off limit of the human eye. The printed image is always superior to the one on the screen.) The command to use then is **fill** . It fills the enclosed area with black. If you want another shade than charcoal black you set

x **setgray**

where *x* is a any real number from *0* (pitch black) to *1* (bleached white). (Negative numbers are treated like zero.) After that the greytone is set at *x* until you change it. (The default setting is always *0* drawing black lines). Thus if you set the grayscale to 1 (white), be sure to revert back to 0(black) as soon as possible, otherwise your text and figures will be invisible, causing you much consternation and confusion.

It is also possible to do color. Color is introduced by a simple command namely

h s b **hsbcolor**

where *h,s,b* are numbers between 0 and 1, where *b* stands for brightness, exactly as in setting the greyscale, and *s* for saturation, while finally *h* stands for hue. Roughly speaking 0 stands for red growing orange into yellow at 0.16 then getting greener, reaching is apogee at 0.3-0.4 then blending into blue at around 0.5 and from then on turning towards violet reaching at 0.7. From 0.8 onwards it is getting more and more red again. A complete color chart is easily programmed, either to be looked at on the sceen or printed on a colorprinter. The drawback of color is of course that it is not easily copied and multiplied.

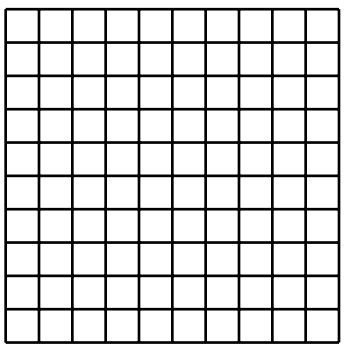
In order to do more serious graphic work, we need to be able to to the usual things in programming, namely recursion, conditionals and doing macros, or subroutines.

4 Recursions

The simplest recursion is of type

0 1 100{ **command** *}* **for**

It produces numbers starting from *0* to *100* on which the chosen **command** is expected to act.



Thus

% ten vertical lines of length 100

newpath *100 10 200{100* **moveto** *0 100* **rlineto** *}* **for stroke**

or

% a grid of one hundred *10 ×10 squares*

newpath *100 10 200{100* **moveto** *0 100* **rlineto** *}* **for** *100 10 200{100* **exch moveto** *100 0* **rlineto** *}* **for stroke**

Or using the trigonometric function **sin cos** which act on

degrees rather than radians we can draw a trigonometric curve

newpath *0 0* **sin** *100* **mul** *0 1 360{* **dup sin** *100* **mul lineto** *}* **for stroke**

This sine-graph will be inconveniently placed, half of it outside the paper, but we will learn to deal with such things later. Note also that the general set-up works equally well for any defined function of one variable (producing just one number). In particular by adding different frequencies we can do graphing of fourier series, a crude example would be something like

newpath *0 0* **sin** *100* **mul** *0 1 360{* **dup dup dup dup sin 100 mul exch** *2* **mul sin** *50* **mul** *3 2* **roll** *3* **mul sin** *30* **mul** *4 3* **roll** *4* **mul sin** *10* **mul add add add lineto** *}* **for stroke**

This clearly is going out of hand and we need to learn how to classify and compactify, which will be the subject of the next section.

5 Macros

This is indispensible, not only are some commands long to write out (like say **newpath**) but often you will use the same chain of commands over and over again.

As to drawing I usually prefer the following set up, which I actually either reconstruct (from memory) whenever I write a new program, or which I (more conveniently) have included in an initializing file (which incidentally should always start with the magic password, so you never need to worry about it). Of course this is by no means canonical, every author should feel to use his own set, but as I have been using them (almost daily?) for many years, they are indeliably engraved in memory, and I will make use freely of it in the text, as otherwise the writing of code would be too tedious. The reader is warned though, that many of the code fragments do not make sense in isolation, but need to be embedded in the context below.

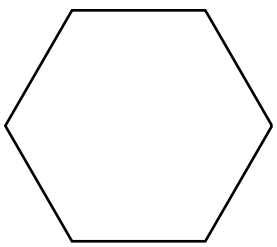
/n *{* **newpath** *}* **def**
/m *{* **moveto** *}* **def**
/rm *{* **rmoveto** *}* **def**
/l *{* **lineto** *}* **def**
/rl *{* **rlineto** *}* **def**
/cp *{* **closepath** *}* **def**
/s *{* **stroke** *}* **def**
/f *{* **fill** *}* **def**

We can also define objects, not just procedures. As examples */pi 3.14159* **def**
/JA (Jockum) **def**
/font /Times-Roman **def**
So we can define the procedue of drawing a line as
/Line *{* **n m l s** *}* **def**

Thus setting from now on
0 0 60 40 **Line**
will produce a line drawn from *(60,40)* to *(0,0)*

(Note that I usually do not bother with defining such simple procedures, they are anyway not part of my usual macro library).

Another procedure is



/c *{* **dup cos exch sin** *}* **def**
Thus
30 **c** $\Longrightarrow$ $\cos(30^{o})$ $\sin(30^{o})$
In particular
n *0* **c m** *0 60 360 {* **c l** *}* **for**
draws a hexagon, and

n *0* **c m** *0 1 360 {* **c l** *}* **for**

draws for all intents and purposes a circle.

Thus defining objects correspond to introducing variables (and fixing their values) in ordinary programming, just like intricate procedures are more than simple notations but corresponds to sub-routines. But the salient feature of PostScript is that there is no need for structuring. New variables and objects do not need to be defined ahead of time but can be introduced whenever convenient, the same thing with subroutines, which can be redefined at whim, and also allows to be arbitrarily nested (in practice of course, if not in theory, there always being hidden limits to all things), however not self-referentially. Thus e.g. if we have defined a procedure **P** say that can be applied to itself (e.g. a matrix operating on pairs of numbers) we are not able to redefine it by **/P** *{* **P P** *}* **def** and we are not able to sneak through the inhibition by denoting **P** by say **P1** with **P1** defined in terms of **P** . However, self-referentiality works well with objects, thus we may with no problem redefine */a {* **a 1 add** *}* **def** . In this way we can easily get around the problem, say with **P** being a matrix, by recursively redefining its numerical entries. (This turned out handy when graphing the fundamental domains of the modular group, which are generated by two basic matrices *S,T*)

One important thing is that when you define a procedure, you may use undefined procedures, it is only when the actual procedure is called up for action, the compiler needs at that moment know everything it calls. Thus it is perfectly workable code to write e.g. the following nonsense

/P1 *{* **P2 Q** *}* **def**
/P2 *{* **A** *}* **def**
/A *{* **pop dup** *}* **def**
/Q *{* **add add** *}* **def**
P1

This means that the writing of PostScript code is very unstructuerd, as already noted above, and in particular this means that you can import code from other programs easily. Also that PostScript codes, like the genetic codes, tends to be unedited, and when used for a long time includes mostly garbage, namely code that has nothing to do whatsoever with the apparent purpose.

However, one note of warning. Unlike subroutines in ordinary programming like 'C', internal variables are not invisible, thus great care has to be taken not to accidentally redefine a globally used variable. Often it does not actually matter, but occasionally some inexplicable bug, which can keep you busy for hours, may be reduced to an accidental slip of using the same letter twice.

6 Conditionals and loops

Certain commands like **eq gt lt** produce as output true or false (which are actually objects on the stack). As examples of true statements are *5 2* **gt** and *-0.1 0* **lt** while false statements are *(5) 5* **eq** or *3 3* **lt** . Depending on the truth-value a certain procedure can be done the syntax being *{* **P** *}* **if** meaning **P** will be executed if following a truth value (uppermost on the stack) otherwise not. (Obviously if this is applied to a stack not topped by a truth-value object, the program will crash, as it always will when asked to do something inappropriate.) A variation, which is almost always used is *{* **P** *} {* **Q** *}* **ifelse** which means of course **P** is executed if true, otherwise **Q** will be executed.

Needless to say truth values can be combined with the obvious commands **or** or **and** thus

x 5 **lt** *x -2* **gt or** *{* **P** *}* **if**

will perform **P** whenever the value *x* satisfies *-2<x<5* .

The recursion previously considered presupposed that we knew in advance how long it would be, if not we have the other alternative of the open-ended loop *{* **P** *}* **loop** which will perform **P** *ad infinitum* . How to stop it? Simply in it implement an exit procedure e.g. *{* **P** *x {* **exit** *}* **if** *}* **loop** . (Where *x* is a truth-value obtained some way or another).

As an illustration we may consider a general ordering procedure. Say that we want to order the numbers 0,1,2 …N_0 in ascending order on a stack with respect to certain numerical values produced by a certain function **T** .

/Order *{* **0 /N 1 def** *{* **/nn N T def** */count 0* **def** *{* **dup T** *nn* **gt** *{ N 1* **roll** */count count 1* **add def** *count N* **eq** *{N* **exit** *}* **if** *}{N* **exit** *}* **ifelse** *}* **loop** */N N 1* **add def** *N N count* **sub roll** *N N0* **gt** *{* **exit** *}* **if** *}* **loop** *}* **def**

The procedure should be clear. Whenever a number *N* is compared with the topmost member of the stack it will either be added on top, in case it is bigger, or the stack is rotated one member for another try. This is repeated up to the number N_0.

7 Scaling and translation

As noted above the standard set up of a rectangle with the lower left hand corner at *(0,0)* and the upper right hand corner at *(600,800)* (approximately, the quotient *y/x* should be close to $\sqrt{2}$) is often inconvenient, but is the default setting at the beginning of each new page (i.e. after the command **showpage** has ended a previous one. To reformate the page we have

x y **translate**

which does the obvious thing, i.e. moving the position of the origin to the point *(x,y)* . Then we have the equally transparent procedure of scaling

a b **scale**

which scales the *x* direction with *a* and the *y* -direction with *b* . Normally we have *a=b* and I usually use the notation *Sc* for the common scalingfactor. Note that those two operations do not commute. When doing a scaling it is important to remember that everything scales, including the text (see below) and the thickness of the lines, thus it is paramount to do something like this

1 Sc div **setlinewidth**

which will produce the standard width of *1* of a line, which is actually default. This incidentally points out the possibility of using different linewidths (something to which we have already alluded). Those are set in the obvious way by the command above.

When doing circles and spheres it is nice to center them in the middle of the paper by *300 400* **translate** and do an appropriate scaling say by */Sc 250* **def** thus being able to work with the normalized radius of one. Incidentally, which is less crucial, but intermittently useful is the *x* **rotate**

which rotates around the origin a given angle *x* given in degree ^{o}.

Finally, although one learns after a while to get a feel for the measurements, they can all be converted to familar centimeters by after each length measure add **cm** in the form of a simple procedure, namely

/cm *{ 72* **div** *2.54* **mul** *}* **def**

Thus

n *3* **cm** *3* **cm** *3* **cm** *0* **rl** *0 3* **cm rl** *-3* **cm** *0* **rl cp s**

does exactly what you think it does. Note that there is no need to add **cm** to 0.

8 Producing text

First we need to initialize a font, and the paradigm is as follows

/Times-Roman **findfont** *30* **scalefont setfont**

Which can be nicely put into a macro like

/F *{ font* **findfont** *sc* **scalefont setfont** *}* **def**

which is convenient if you plan to do frequent changes of font and scales.

Thus we can do e.g.

/font /Helvetica-Bold **def** */sc 20* **def F**

Note that this is a fairly big font, the letters of this page correspond to about 12. (If we have done a significant scaling by say *Sc* it is imperative to scale down *sc* simply by */sc 20 Sc* **div def** otherwise you will be up for an unpleasant surprise.)

To print something out we need a string co-ordinates and where it should be put thus, after we have initialized some font we can write

(Jockum) 100 700 **m show**

Which will display 'Jockum' in the left upperhand corner (unless we have done some scaling and translating beforehand)

The following will give the strings printed after each other

(Jockum) 100 700 **m show** *(is)* **show** *(a)* **show** *(nice)* **show** *(boy) show*

Jockumisaniceboy

thus as on the left

To avoid it we can either insert spaces within the strings

(Jockum) 100 700 **m show** *(is)* **show (a) show** *(nice)* **show** *(boy) show*

Or to do

/mm *{* **4 0 rm** *}* **def**

(Jockum) 100 700 **m show mm** *(is)* **show mm (a) show mm** *(nice)* **show mm** *(boy) show*

Clearly we can if we prefer absorb **mm** into a new definition say of **showm**

To do a centering of text we can use the following routine

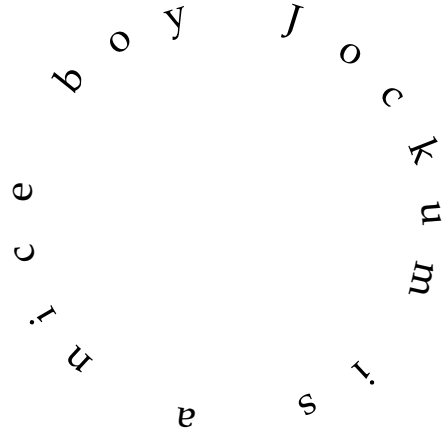
/cshow *{* **dup stringwidth pop** *-0.5* **mul** *300* **add** *3 2* **roll m show** *}* **def**

Which works on a stack

700 (Jockum is a nice boy) **cshow**

A more frivolous exercise would be the following

/Sc 100 **def** *300 350* **translate**



Sc Sc **scale**

/font /Palatino-Roman **def** */sc 30 Sc* **div def F**

( )(y)(o)(b)( )(e)(c)(i)(n)( )(a)( )(s)(i)( )(m)(u)(k)(c)(o)(J) 1 1 21{ **pop** *0 1.2* **m show** *360 21* **div neg rotate** *} for*

One should also keep in mind that when you write a parenthesis it is best to write \050 and \051 respectively because expressions like *())* greatly confuses PostScript. Thus if you want to write *((mean boy))* the string *(\050mean boy \051)* is to prefered, although in this particular instance PostScript would be able to handle matters as the parenthesis are locally

nested.

Another useful thing to know is the possibility of converting numbers to strings (invaluable if you want to do some hunting down of recaltricant bugs). In order to do so we need to prefigure by a

/str *20* **string def**

Then

15 **str cvs** $\Longrightarrow$ *(15)*

which then can be printed out.

We can also do some playing-around, like shading letters and such things. More interesting though is the possibility of doing so called blackboardbolding. Here is a simply routine

/bold *{* **dup dup show stringwidth pop** */ss* **exch def** *-0.8 ss* **mul** *0* **rm show** *}* **def**

where the parameter *-0.8* is a matter of taste and experimentation. Then we can go ahead and do

(R) 100 500 **m bold**

Finally we would also find it useful to print out Greek letters and mathematical symbols. For this there is the font */Symbol* whose coding is to be found in the appendix. Suffices it to point out that *(\160)* corresponds to π.

9 Arrays and such things

Sometimes it could be handy to stock the contents of a stack in memory. The object that handles such things is the array. We can either define one by hand, like */Arr [(2) 3 /Times-Roman]* **def**

which places three objects in the array *Arr* . In fact we can even insert things like *5 3* **mod** because this after all produces a number *2* or truth-values like *3 2* **gt** into an array. If we want to take out an object we simply do

Arr 2 **get** $\Longrightarrow$ */Times-Roman*

Thus, not surprisingly the objects are counted from zero and up.

We can also define an array as

/Arr 10 **array def**

which we now can start to fill by the **put** command. Thus

Arr 2 (Boel) **put**

puts in position two the string (Boel) and actually replaces the object */Times-Roman* **in case we apply it to the old** *Arr* . There are also commands that puts on the stack all the objects of an array along with the array itself. Such routines can easily be implemented and there is no reason to load your memory with such things.

Just as there are limits to the depths of stacks (too long and there will be overflows) thus in programs that work with many steps it is a good thing to continually clean out the stack with the judicious use of **pop** , there are also of course limits on the capacities of arrays, the exact numbers probably differing for different implementations and something the system-man should be able to anser, but most likely will not. (Anyway experimentations may reveal the actual limits.)

10 3D-pictures

Points in space are given by 3 coordinates (i.e. three numbers x y z in the stack). By suppressing the last (i.e. doing **pop**) we simply project it to the x-y plane. This is a rather stupid thing to do unless we do some rotations of the plane of projection. To do so we need to construct some routines.

/dup2 *{* **dup** *3 2* **roll dup** *4 1* **roll exch** *}* **def** *% xy to xyxy*

/add2 *{* **exch** *4 1* **roll mul** *3 1* **roll mul add** *}* **def** *%abxy to ax+by*

%x y t to xcost-ysint xsint+ycost

/vrid *{* **dup cos exch sin exch dup2 exch neg** *6 4* **roll dup2** *6 2* **roll add2** *5 1* **roll add2** *}* **def**

/vridx {vrid} def

/vridy *{3 2* **roll** *4 1* **roll vrid** *3 1* **roll exch** *3 2* **roll** *}* **def**

/vridz *{4 2* **roll** *3 2* **roll vrid** *3 2* **roll** *}* **def**

The reader who finds the mental arithmetic of stack manipulation tedious may 'cheat' by simply defining the handy

/ed *{* **exch def** *}* **def**

and then do /z **ed** */y* **ed** */x* **ed**

translating the contents of a stack into simple variables x,y,z and then do straightforward manipulations on them. (However, the advantage of the stack yoga is that one does not introduce new variables, which may in fact be old and interfere with the process)

We may then draw a perspective of a cube by simply specifying its vertices (say by combinations of 1,-1) and draw the appropriate edges, one of which would be

newpath *1 1 1* **U pop moveto** *1 1 -1* **U pop lineto stroke**

Where **U** *would specify some particular rotations (as a combination of different* **vrid** *). The reader who would find the writing down of the twelve lines odious could think up some way of generating the eight vertices and deciding when two vertices should be joined by an edge. Here is a suggestion.*

%generating vertices

/V *{ /v* **ed** *1 1 3 {* **pop** *v 2* **mod** *0* **eq** *{1}{-1}* **ifelse /v v 2 idiv def** *}* **for** *}* **def**

%making variables (this is really cheating)

/XYZ *{ /z1* **ed** */y1* **ed** */x1* **ed** */z* **ed** */y* **ed** */x* **ed** *}* **def**

%computing distance between two points

/dist *{x x1* **sub dup mul** *y y1* **sub dup mul** *z z1* **sub dup mul add add** *}* **def**

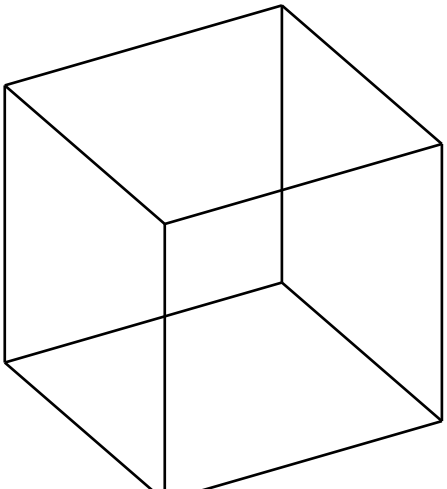
%drawing a line

/L {n *x y z* **U pop m** *x1 y1 z1* **U pop l s} def**

We are then ready to set it up

cube *{0 1 7{/v1* **ed** *0 1 v1{/w* **ed** *v* **V** *w* **V XYZ dist** *4* **eq** *{* **L** *}* **if** *}* **for** *}* **for** *}* **def**

(Why did we use *v1* instead of the simpler *v* ?) The picture will be as on the left



Now we may want to show a non-transparent cube. This means that we are going to hide lines. This is naturally effected by the filling out of polygons, and thus the strategy is to first paint the most distant polygons and then proceed to the closest. One measure of distance is to look at the distance to the center of gravity of each polygon. (Clearly if we are going to have many polygons we need to subdivide them in smaller ones in order not to get unplanned surprises, but in the simple situation of a

cube we need not worry about it. Now we have before defined an ordering routine which we can use. Thus a suggestion for a program would be the following.

/face *{* **n** *-1 -1* **A m** *-1 1* **A l** *1 1* **A l** *1 -1* **A l cp z** *}* **def**

/A *{1* **V U pop** *}* **def**

%The following generates the faces of the cube

/Vd *{* **dup** *4* **lt** *{ 90* **mul** */v* **ed /V** *{v* **vridy** *}* **def** *}{2* **mul neg** *9* **add** *90* **mul** */v* **ed /V** *{v* **vrid** *}* **def** *}* **ifelse** *}* **def**

%This function is used for ordering the faces

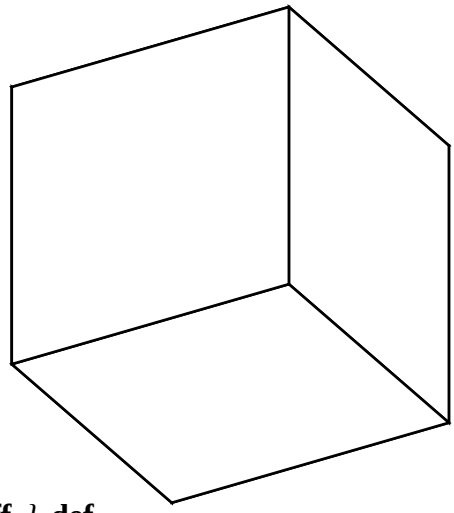
/T *{* **Vd** *0 0 1* **V U** *3 1* **roll pop pop neg** *}* **def** */N0 5* **def**

Order

/cc 0 **def**

{ **Vd** *1* **setgray /z** *{* **f** *} def face 0* **setgray /z** *{* **s** *}* **def face** */cc cc 1* **add def** *cc 5* **gt** *{* **exit** *}* **if** *}* **loop**

The result can be seen below . A simpler case of



actually using the technique of hiding is given by the graphing of a function, when the angle **U** is chosen with some care, as to be able to predict beforehand the ordering. So let us consider the graph over a rectangle in which we are going to graph the most distant first. If the corners of the square are given by ±1 ±1 and the grid has mesh *d* we may set up the following

/d 0.1 **def**

/da *{d* **add** *}* **def**

/Ff *{* **Fun U pop** *}* **def**

/ff *{* **n** *xx yy* **Ff m** *xx* **da** *yy* **Ff l** *xx* **da** *yy* **da Ff l** *xx yy* **da Ff l cp z** *}* **def**

/Box *{1* **setgray /z** *{* **f** *}* **def ff** *0* **setgray /z** *{* **s** *}* **def ff** *}* **def**

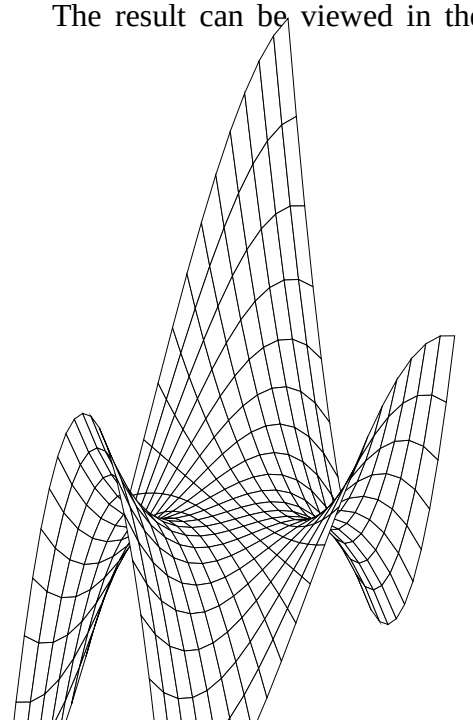
/Fun *{* **dup** */yf* **ed exch dup** */xf* **ed exch function** *}* **def**

1 d **neg** *-1 {/yy* **ed** *-1 d 1 {/xx* **ed Box** *}* **for** *}* **for**

Where we can use the viewing angle gioven by **/U** *{30* **vridz** *60* **vridx** *}* **def** . It only remains to choose the function one example may be the Monkey-saddle, i.e. the harmonic polynomial given by the real part of z^3 i.e the function below

/function *{ xf xf xf* **mul mul** *3 xf yf yf* **mul mul mul neg add** *}* **def**

The result can be viewed in the picture



is invited to experiment with other choices of functions.

We may also 'jazz' up the function graph, e.g. we can paint every other square black (for some inscrutable reason), or being more sophisticated introducing a light-source and thus shading each square appropriately. (Simply computing the cosine between the normal and the direction of the light, through the standard inner product), There is a serious problem though. It is hard (although not in principle impossible) to determine which rectangles are hidden from the light of the light-source, because unlike us, the code is blind. We have no difficulty from our vantage point by simply looking. In the case of spheres it has a simple solution a special but important case of 3-D graphing that justifies its own section.

11 The Sphere

The sphere is a useful and easy thing to implement. It can be done in a variety of different ways. One obvious entry into the possibilities is to recall the procedure **/c** *{dup* **cos exch sin** *}* **def** which defined the circle of radius one by

/circle *{* **n** *0* **c m** *0 1 360 {* **c l** *}* **for** *}* **def**

With some obvious modifications we can make this circle a latitude at height *h* namely do

/c *{dup* **cos** *r* **mul exch sin** *r* **mul** *}* **def**

/circle *{* **n** *0* **c** *h* **U pop m** *0 1 360 {* **c** *h* **U pop l** *}* **for** *}* **def**

Where we put */r h h* **mul neg** *1* **add sqrt def** and then do a sequence of different *h* , the details by now safely left to the reader. Note that *h=0* corresponds to the equator. The command **U** is clearly defined, as usual, by a short sequence of rotations, determining from which angle we aree viewing the sphere.

This taking care of the latitudes. A longitude can be thought of the equator rotated arund some axis in the *x-y* plane, say the x-axis. Thus

/circle *{* **n** *0* **c** *0 90* **vrid U pop m** *0 1 360 {* **c** *0 90* **vrid U pop l** *}* **for** *}* **def**

To get the other longitudes we simply rotate around the z-axis, thus

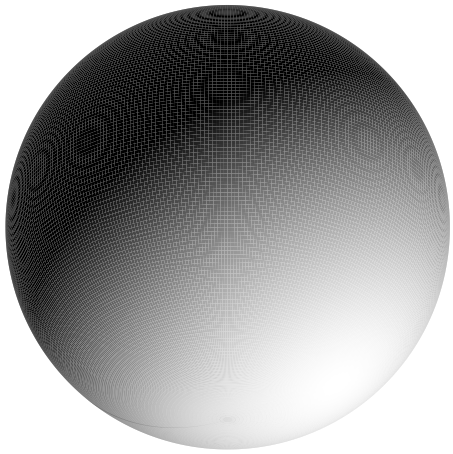
/circle *{* **n** *0* **c** *0 90* **vrid** *ang* **vridz U pop m** *0 1 360 {* **c** *0 90* **vrid** *ang* **vridz U pop l** *}* **for** *}* **def**

for different angles *ang* .

Now if we do not want a transparent sphere but an opaque one, certain lines will be hidden, but which ones? For spheres the answer is wonderfully simple. Looking at a sphere from infinity, the horizon coincides with a great circle cutting the sphere in half. Only the positive half can be seen, i.e. those with positive *z* -value. Thus we simply replace the fragment **U pop l** by **U** *0* **gt** *{* **l** *}{* **m** *}* **ifelse** and we present only the visible half. If we are doing a real perspective show, being located at a finite distance, there is a slight modification to be done, safely left to the reader.

The ease with which opaqueness can be handled for spheres should be contrasted with the difficulty for ellipsoids, even rotationally invariant such. It can of course be figured out mathematically and implemented but it is not entirely trivial, illustrating one of the nice features of PostScript programming, giving you elementary, yet intriguing, mathematical problems to solve, rather than leafing through a manual.

There are of course other ways of producing spheres, one is to use the spherical co-ordinates as given by the latitudial and longitudial grid directly exploiting the standard formulas to give the 3-dimensional *x,y,z* co-ordinates which are easily implemented. The figure below plots the visible longitudal/latitudal rectangles shaded by a light-source not coinciding with the direction of the observer. The shading is simply given by setting the greyscale equal to the *z* -value under **V** (the direction of the light source) noticing that *z* **setgray** returns black also for negative values of *z* .



As is well-known we can to each skew 3 ×3 matrix *M* associate an element exp(M) of

SO(3) whose axis of rotation is given by the kernel of M **.** This can be done simply by the following formula, (in which M has been normalized so that $Tr(M^2)=-2$

$\exp(\theta M)=I+\sin\theta M+(1-\cos\theta)M^2$

and hence be straightforwardly be implemented in PostScript. This has been done, and in fact it gives a nice routine for drawing the great circle arc between two given points on the sphere, and thus in particular to draw spherical triangles.

Finally no treatment of the sphere is complete without some mention of the presentation of map-projections. In principle a map-projection is given by a function $F(\theta, \varphi)$ of the spherical co-ordinates, and once the function has been implemented it should be straightforward, once we have documented polygons of the major connected components of the landmasses. There are, however, some snags. Sometimes we have the problem of monodromy, as when we have a cylindrical projection, like Mercator, or only certain parts of the globe is visible, like in the orthographic (the earth seen from infinitely afar) or the gnomic, when only half of the earth is visible. In the first case we can simply do hidden lines, but if we want to fill out polygons, we run into problem. Then the hidden point, normally to be mapped inside the circumference of the (circular) horizon, is pushed onto the boundary. This map is only well-defined outside the center, and for points close to the center, the procedure will result in ugly results. Needless to say such things can be taken care of by a variety of methods, but it once again illustrates how mathematical phenomena (like retraction to the boundary) can be made tangible in PostScript programming.

12 Implementing PostScript in tex

It is possible to insert PostScript pictures into tex-documents. At least it works in amstex using an input of \epsf. The commands used are \epsfysize, giving the vertical size of the picture and the calling procedure \epsfbox{*.ps} inserting the address of the file. For this to work reasonably well one needs to supply the code with so called bounding boxes, i.e. giving the corners of a rectangle in which the picture resides. Without it, it usually works nevertheless, but sometimes unpredictably. This has forced me to design my own software to put pictures exactly where I want them.

13 Final advice

Experiment! There is no substitute for hands-on experience.