The Plot-Files

C. Godsalve
email: seagods@hotmail.com

September 5, 2009

Contents

1 Introduction

So what are the "*Plot-Files*". You will not be surprised to find that they are for plotting graphs and visualising data. There is a lot of software "out there" to do this kind of thing of course.

However, I couldn't get along with some of them, could not afford others, and at the same time was learning about *OpenGL*. (That is the Open Graphics Library.) The result was a self imposed project — the *Plot-Files*.

There are different ways to plan a "graphics package". For instance one might have a unified windows based based point and click system. We do not go down this route. Instead we have a series of stand alone programs designed to be run from a terminal (they can be run by clicking on an icon) and these read a pre-existing data file written in a format for that individual program to read. I developed these programs on a *linux* operating system. Hopefully, you will be be able to use them if you have a non-linux operating system. This may take some work to adapt to Mac OS-X or Windows, and I cannot guarantee anything.

At any rate, I shall write as if these programs were run on a *linux* or $UNIX$ system. Symbolic links can be set to access them from anywhere. They were designed to meet my own current needs, and so do not include everything you might want. They may be expanded and improved upon in the future.

So what are these stand alone programs? What is their current status? The first and simplest is $PlotIt$. This plots multiple curves which are functions of a single variable such as $y = f(x)$ on the same axes. Next we have $PlotIt2$. This plots multiple curves in three dimensions: that is we have functions such as $z = f(x(t), y(t))$ where t is some parameter.

Next we have $PlotCont$, which plots a (single) function of two variables such as $z = f(x, y)$ as a flat coloured map with contours. Next again we have $PlotSurf$. This plots multiple functions of two variables as a two dimensional curved surface embedded in a flat three dimensional space. Moving on, we have $PlotVol$. This is a generalisation of $PlotCont$. Here, we can visualise a function $w = f(x, y, z)$ as two dimensional curved surface contours (isosurfaces) in a three dimensional space: or if you will a three dimensional curved surface embedded in a fourth dimension.

Alongside these there is $PlotPol$. A polar coordinates version of $PlotCont$. This is *not* a generalised polar plot. It is specifically written for plots on a hemisphere such as might be (for instance) needed to visualise the radiance of skylight as a function of direction. The radius variable is therefore tied to a zenith angle between 0 and $\pi/2$ radians. Curves can be plotted over the colour/contour map (which, for instance, may be satellite tracks). Also, we have two symbols which can be plotted so as to represent (for instance) individual solar and satellite positions.

Lastly, we have two programs utilising World Vector Shore line data. One is $PlotWorld$, a variation on $PlotCont$, which plots a world map on top of the colour/contour map. Finally, there is $PlotGlobe$. So far this is only a place-holder. It wraps a bitmap image of the Earth around a sphere and plots the shoreline over the bitmap. This will be adapted later for different purposes.

All these $Plot - Files$ use the same OpenGL "camera" functions. You can use arrow keys to move left and right, up and down, in and out in $PlotIt$ for instance. When Using $PlotVol$ you can use the arrow keys to move spin the object round in three dimensions as well as zooming in and out. The mouse can also be used to change the camera view

direction, or if the left mouse button is held down, you can change the camera position. The details of the full camera functionality are explained in the camera.pdf file at this site. There is also an mpeg video of how these function using a landscape as an example. Yes, you can do "victory rolls" and "loop the loops" in flight mode.

If you wonder if the name $Plot-Files$ and the reference to "out there" has anything to do with the "X-Files", then you are correct! That is to say there are a lot of functions and header file definitions common to all the programs. These are stored separately in a directory called $XFiles$. Why the $XFiles$? Well lets just say that some of the most "spooky" debugging took place here.

Lastly we come to the status of these codes. I shall apologise in advance to anyone who tries to follow the actual code. I started programing by teaching myself $BASIC$ on a Sinclair ZX81 more than two decades ago. Then I taught myself $FORTRAN$ during my Ph.D., and I have never shaken off my bad habits as far as writing code goes. My C++ is also "teach yourself in 24 hours" stuff. I am by no means a professional programmer. If you were to ask what my programming style was I would say "Heath Robinson Hacker". (Heath Robinson was a British Cartoonist famous for drawing wildly eccentric machinery.)

I should say that this is not even an alpha-zero release. If it works for you that's fine. If it doesn't, tell me about it and I shall try and get round to fixing it as soon as time allows. I say it is a pre-alpha release because I have not tested everything extensively as yet, and I dare say that some problems might show up from time to time.

The known bugs are as follows. For postscript output, we have used gl2ps. (The URL for this is noted in the $PlotItsection$. Whether it is my fault or not, I don't know, but stippled lines do not appear as they do on the screen, and surfaces which are semi-transparent on the screen come out as solid when viewed as postscript. For large complex surfaces postscript output is *very* slow. The program hasn't frozen, just go and have a cup of coffee or something until it has finished the output. Mostly, the postscript output is quick though.

We also have bitmap output. This is $Screenshot$ and is taken straight from a tutorial program by Ben Humphries who I shall give a further mention of. If you have X-Windows you can make mpeg or ogv videos of your surfaces as you spin around and zoom in and out. The $Plot-Files$ themselves do not output mpeg, but you can do this via $XVidcap$ or $gtk-recordmydesktop$. If you don't have them, you can download them from http://xvidcap.sourceforge.net and http.recordmydesktop.sourceforge.net.

Waiting for things to be completely perfected can mean waiting forever. So, I am making this sub-alpha release now.

This particular document is just a brief introduction. At this site I have written articles on the $Camera$ class with a detailed discussion on the various camera controls. There is also a more in depth discussion of $PlotVol$ and $PlotSurf$. Also at this site are links to

short video tutorials on how to use these programs. In fact, these may be preferable to this article for some readers.

At this point I shall mention that I am indebted to DigiBen's tutorials (Ben Humphries is at http://www.gametutorials.com) which these programs grew out of. He has been kind enough to grant permission for me to distribute the chunks of code such as the SDL initialisation, bitmap readers, and the Screenshot (which I adapted for 64 bit machines). This is because I do not use any of his loaders or his octree stuff. (The octree code at this site is entirely my own.)

Also I am indebted to other tutorials at NeHe Productions (http://nehe.gamedev.net). Then more tutorials available at the OpenGL and SDL sites (http://www.opengl.org and http://www.libsdl.org).

2 *Unpacking the Plot-Files*

I can only write about how you do this on a *linux* operating system. That is not to say that you have to have *linux* for this to work. In fact, if you get this working on $MicroSoftWindows$ or Mac OSX, or even $UNIX$, email me so I can put instructions here (alongside an acknowledgment of course). Before anything else, you *must* have a C++ compiler. If you have a *linux* system then you will have *gcc* but might not have installed it. You can get gcc for other systems from http://gcc.gnu.org.

First, make a directory, we shall call it $Fred$ for the current purposes. Inside that, make another directory called "Graphics", and unpack all those *Plot* directories here. Then unpack the $XFiles$ as a subdirectory of $Fred$. You need to have $OpenGL$ and SDL. You also need to have $true-type$ installed, as well as *gltt*. On some *linux* distributions, these might be on the install discs but not automatically installed.

If these are not on your system you can go to my pages http://seagods.stormpages.com and follow the links to "Getting Started With OpenGL and SDL" and download them. Note, my instructions on compiling these don't work with the GCC-4 series. They do compile and work with the last of the GCC-3 series.

Now, in the $XFiles$ look at $TrueType.h$. You will see lots of statements with "#include". Find out where these things are on your machine. A bit lower down and you will see things like "font1.open". Again, if necessary, change these to the location on your machine.

Now, go to "Fred/Graphics/PlotIt". You will see a file called "Plotit.h". In here, you will see things other "#include" statements for GL and SDL. Make sure these are right for your machine. (Of course the same goes for all the $Plot-Files$.) Now, do *makeclean*, and then *make*. (Or make sure that "compile" is executable, and then do *compile*.) If all

has gone well, you now have an executable file *PlotIt*.

3 *PlotIt*

As I mentioned, *PlotIt* needs a data file in *exactly* the format it requires. Otherwise BANG! There are no safety belts. You will see a small file called data.dat. This is just a tiny "joke" data file. You will see that the first line *must* consist of just one integer (no decimal points). This is the number of curves (*nplots*) you want to plot. In this case it is three.

The next line *must*consist of five integers, separated by whitespace. The first number is *ndata*. That is the number of (x, y) point pairs. Again, the following *ndata* lines *must* be these x and y values separated by whitespace. But, back to that line of five integers. The second integer is *ntype*, it must be zero if you want to plot lines, and 1 if you want to plot points. (I have not coded up for any type of symbols or error bars as yet.)

The next number is *ncol*, which determines the *initial* colour of the line or the points. These can be white, black, dark red, bright red, dark green, bright green, dark blue, bright blue, orange(ish), bright yellow, and blue-green. The colour white is zero and black is one, and the blue-green is 9. The colours can be changed once *PlotIt* is running.

Next we have *nstyle* which is the line type. This must be there even if we are plotting the just the data points. This must be 0,1,2,3 or 4. The number zero is a solid line. The others are stippled. Finally there is *npoint* which is the point size or line thickness. Set it to 1 for a thin line (or a small point).

This is the example data file *data.dat*. If your data is in "Joe.data" all you need to do is type "PlotIt Joe.data ¡Enter¿" at the command line. If "Joe.data" is missing, *PlotIt* will look for its default data file name which is just "data.dat".

```
3
3 0 5 0 1
0.1  0.06
0.2  .14
0.3  .20
4 0 6 1 2
0.0  -.1
0.5  .1
1.0  .1
1.2  0.07
5 0 7 2 3
-.1  -.2
.4  -.1
```

```
.6  -.2
.8  -.1
1.0 0.2
1   1
x@values   per@cm
1   1
y@values   per@year
1
Fred@Jones
Harry@Red
Mary@Green
```

So, we have 3 curves to plot (line one). Then the five numbers discussed above. The value of *ndata* is three, and we have three data pairs. Then we have another line of five integers with $ndata = 4$, followed by four lines of data points, then the last line of five integers for the last of the curves, with $ndata = 5$. Then of course we have five lines of (x, y) pairs.

After this we see a line with two integers. The first must be 1 if there is an x-axis label, and zero otherwise. The next is 1 if there are units to go with the x axis label and zero otherwise. (Don't put 0 1.) We want an x-axis label "x values". To achieve this we must replace any space with an "@", hence x@values. The same goes for the units, we have chosen "per cm". If we do not want an axis label, and put "0 0" instead of "1 1" for the integers *ixtext* and *ixunits*, the next line *must* be the integers *iytext* and *iyunits*. (*There must be no blank lines anywhere.*) So, after "x@values per@cm" we need the values of *iytext* and *iyunits*. These are both one, so the line is followed by the text for these.

We are almost there. We may, or may not want to have a *legend* to explain what the different curves are. If we want the legend we must have a line of text for each curve. The example data gives the plot in Fig.1.

Now we have a window with the plot of our three curves, we can do things like change line colours and thickness, make changes in text positions and so on. To find out how to do these things, all you need to do is press $< F1 >$. This will give a help screen. Pressing $< F1 >$ again gets rid of the help screen. Pressing ¡Esc¿ quits the program. The help screen is exactly the same for all of the $PlotFiles$. This means that not all the controls will work. For instance, $PlotIt$ does not use OpenGL lighting. So if you press $< F6 >$ (change light position) nothing will happen. $< F11 >$ and $< F12 >$ give you bitmap or postscript output. The arrow keys modified by ¡Ctrl¿ will move the plot left and right or up and down. The up and down arrow keys on their own will let you zoom in and out.

4 $PlotIt2$

This program plots "space curves" $(x(t), y(t), z(t))$. It comes with a small program called *datawrite* which gives an example of how the data must be written. To compile it all you type is "g++ -o datawrite datawrite.cpp". The example output should be something like Fig.2. Like all the PlotFiles,you just type "PlotX filename" (where X=It2 in this case). The default is "data.dat" as in $PlotIt$.

The output in Fig.2 is a postscript file written by $gl2ps$. Unlike the bitmap output of Fig.1, it can be scaled without loss of quality. We mention a few things which are common to all the 3D plots. First, the z axis text and numbers rotate with you as you rotate the camera, and they only appear on the two outermost of the four z axes. The other thing is that the data is mapped onto a cube. That is if x and y vary between 0 and 10, and the z data vary between 0 and 0.1, the z axis appears to have the same dimensions as the x and y data. I have not included the option of leaving the axes unscaled. The viewer has to read the numbers to know the scales. At present, I have no intention of changing this. The same goes for all the $PlotFiles$.

5 $PlotCont$, $PlotPol$, and $PlotWorld$.

These programs are similar in that they all serve the purposes of visualising single functions of two variables. The function values are colour coded between the maximum and minimum function values so that red is close to the maximum and blue is close to the minimum. (The user has no control over this.) The plot is a flat coloured surface, and there is a legend rather than a z axis. To aid the visualisation, contours are plotted at all the function values that would correspond to the minor tick marks on a z axis if we had one (as in $SurfPlot$ which we shall see later).

$PlotWorld$ is a specialised version of $PlotCont$ where the data must be between $\pm 180^{c}irc$

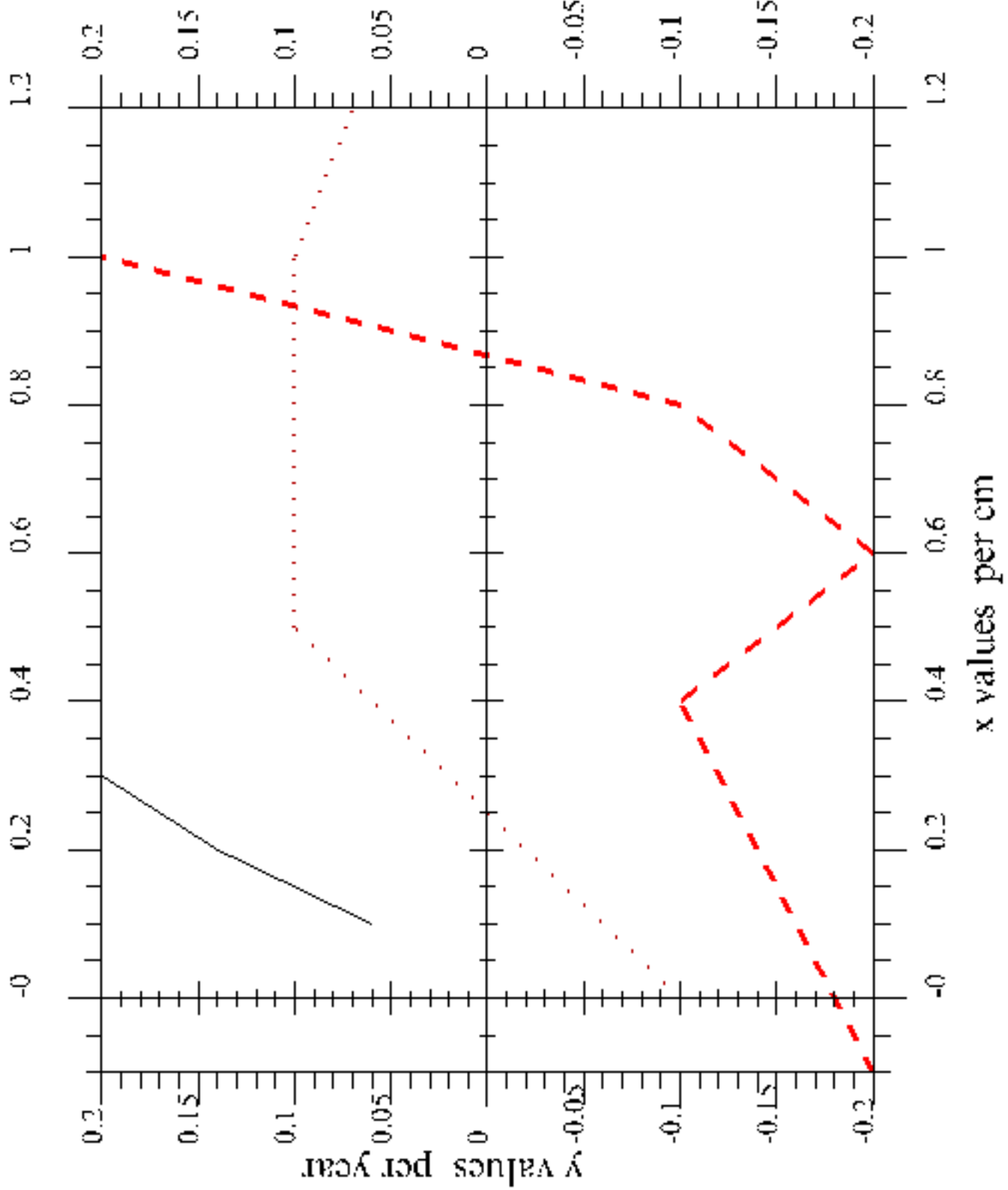

Figure 1: The output of *PlotIt* for the example "data.dat"

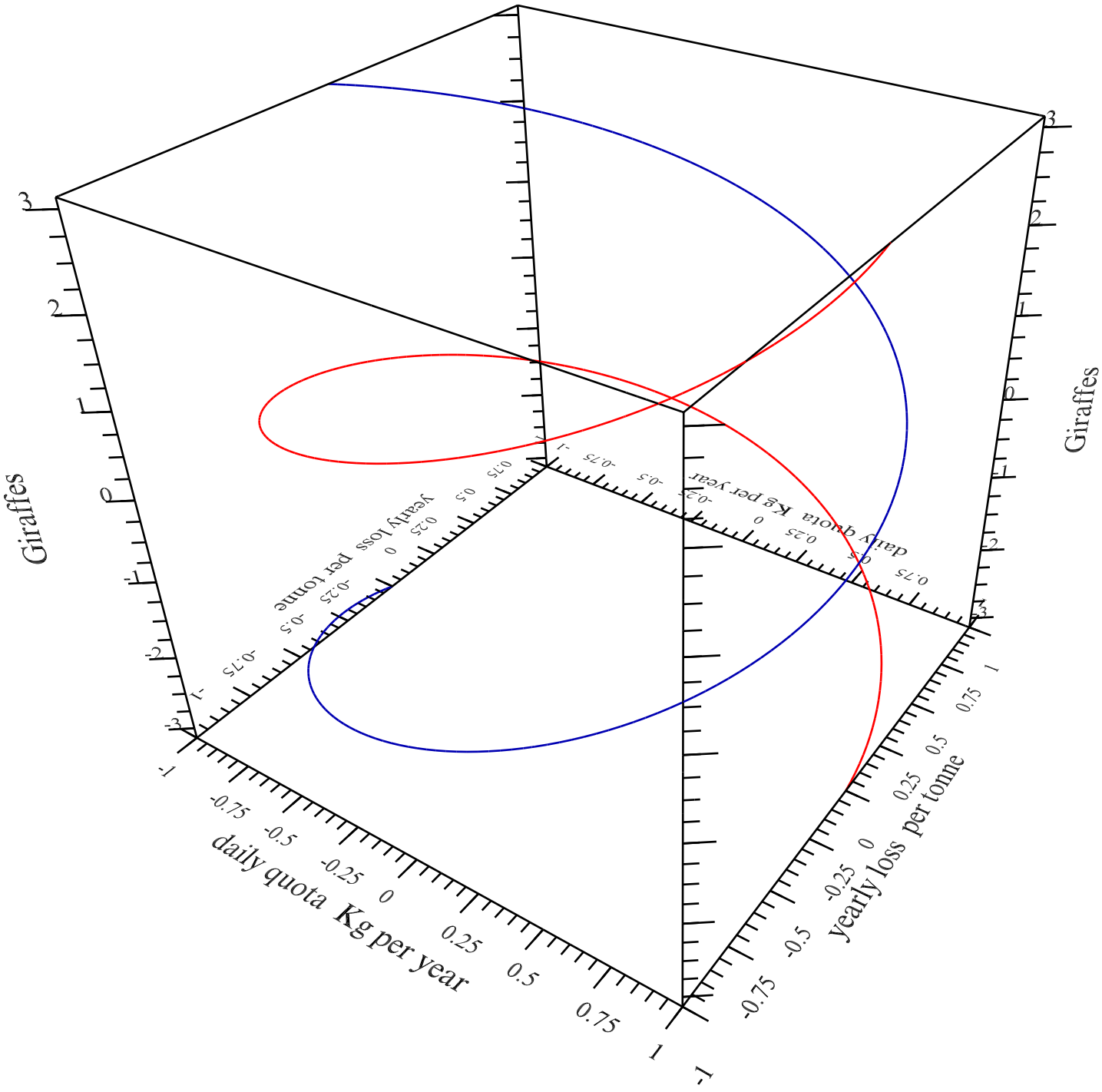

Figure 2: Two helices as visualised by *PlotIt*2 for the example "data.dat"

in longitude and $\pm 90^{c}irc$ in latitude. We overlay a world map (Mercator projection) obtained via http://rimmer.noaa.gov. (Note this leaves an odd gap or so in various coastlines, however this should not be noticeable). Also, PlotPol is a polar plot, but it is specialised in the sense that the radius is meant to represent a zenith angle between 0 and 2π.

The data files all have to have the following format. The first line is to consist of two integers *ndata* and *ntri*. In the example file these are 496 and 900. That is we have 496 data triplets (x, y, z) representing nodes numbered from 0 to 495, and 900 triangles numbered 0 to 899. After this first line, we have 900 lines each consisting of the (integer) node numbers of a triangle. Then we have the (x, y, z) triplets (one triplet per line), and axis text just as we had in *PlotIt*. Each of these directories has an example *datawrite* file. Note however, that *any* kind of triangulation generated by any program. So we might have, for instance, temperature data generated by a 2-D finite element code as the input. In addition *PlotPol* can overlay the contour plots with curves (satellite tracks) and symbols (Sun and satellite positions). We have overlaid an Archimedian spiral and a polar rose which are obviously nothing like satellite tracks!

Figures 3, 4 and 5 are example outputs from *PlotCont*, *PlotWorld*, and *PlotPol*.

6 *PlotSurf*

In Fig.6, we see the bitmap output for the example *PlotSurf* data files. This time we the visualistation is of two functions of two variables as curved surfaces. The defined is defined on an arbitrary triangular mesh. Though the bitmap conversion to postscript has seen some degradation, we can see the use of transparency. As with the three *PlotFiles* seen in the last section, we can toggle whether the surface grid is plotted, and whether the contours are plotted (by pressing 'g' or 'c'). If the user does not wish to use transparency, pressing 'b' toggles the blending that allows transparency.

7 *PlotVol*

Finally, we have reached the last (at present) of the *PlotFiles*. This program plots isosurfaces of a function $w = f(x, y, z)$. The input data have a similar form to that of *PlotCont*, except now the data must be a tetrahedral mesh rather than a triangular one. The *basiccube* program provides a simple tetrahedral mesh. However, the mesh is in general an arbitrary tetrahedral mesh and might be, for example, generated by a three dimensional. At the time of writing, I haven't yet got round to fixing periodic boundary conditions. As with *PlotSurf* we can change the light position and colour, but the isosurface colours are predetermined by the "main tickmarks" of the w axis which is represented through a legend.

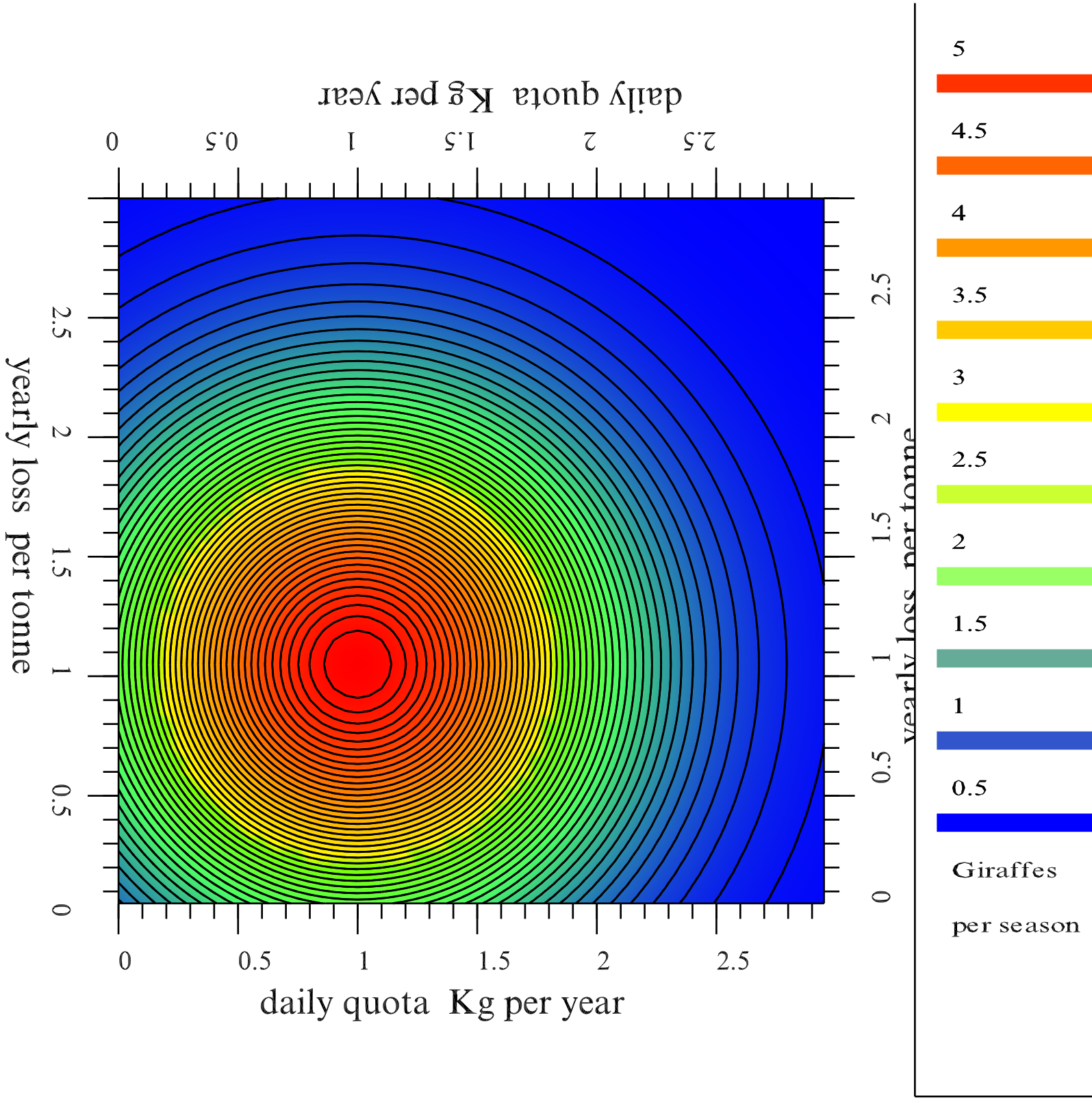

Figure 3: A contour/colour representation of a function of two variables from *PlotCont* for the example "data.dat file". The legend has been dragged to the left using the mouse by holding down the "left clicker". The artifacts on the screen do not appear when printed.

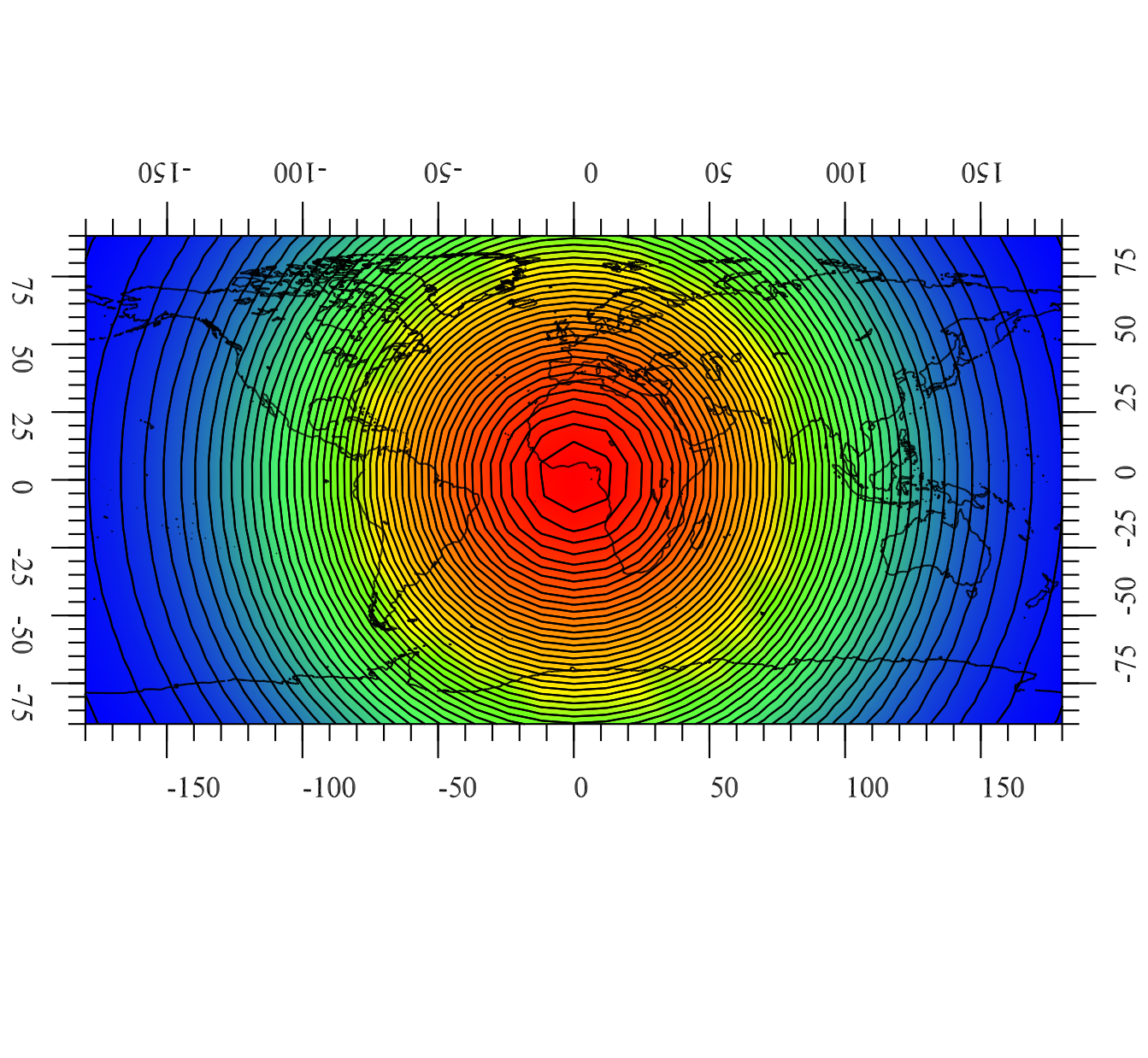

Figure 4: A contour/colour representation of a function of longitude and latitude from *PlotWorld*

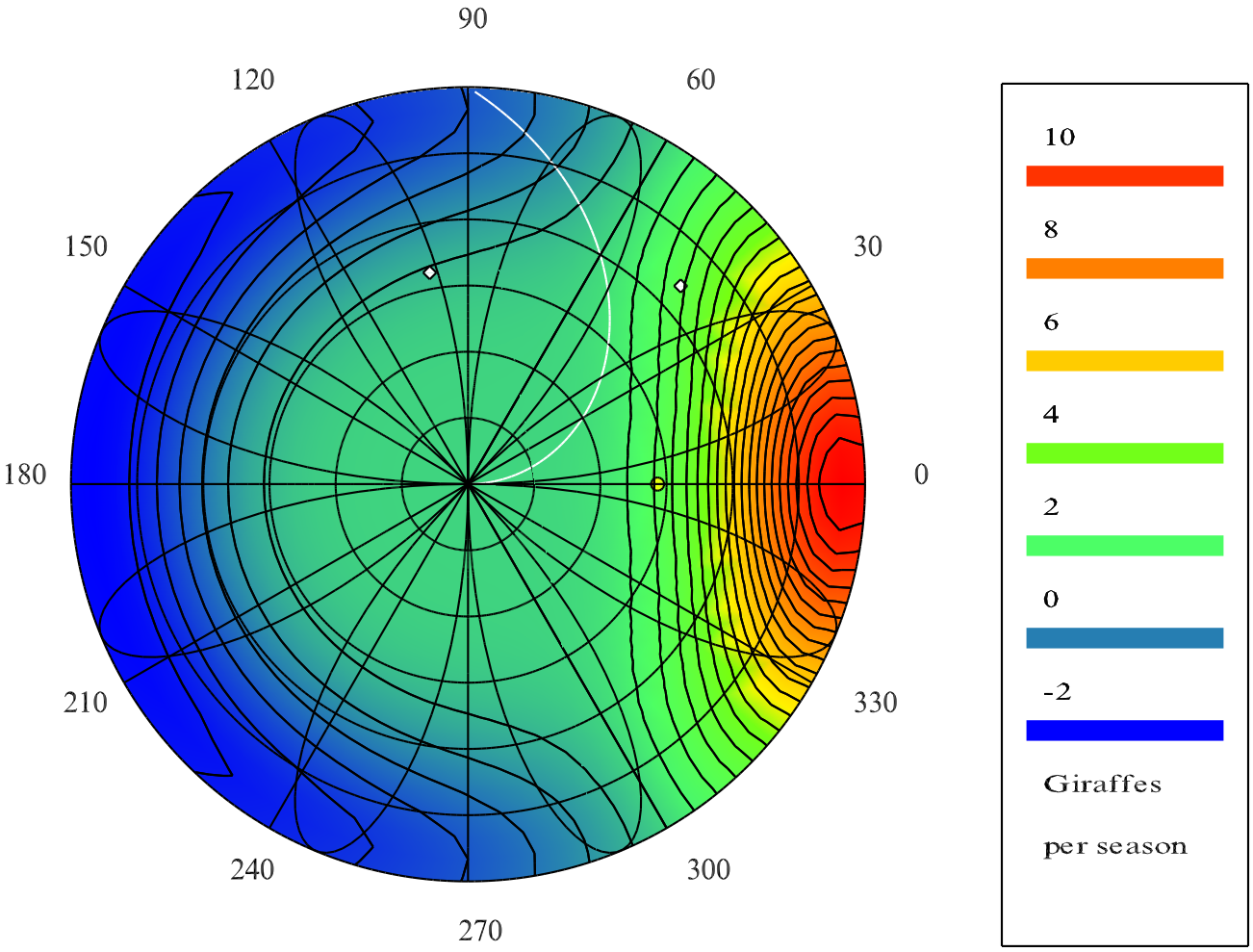

Figure 5: A contour/colour representation of a function of zenith angle and the angle from the Principle Polar Plane from example data for *PlotWorld*

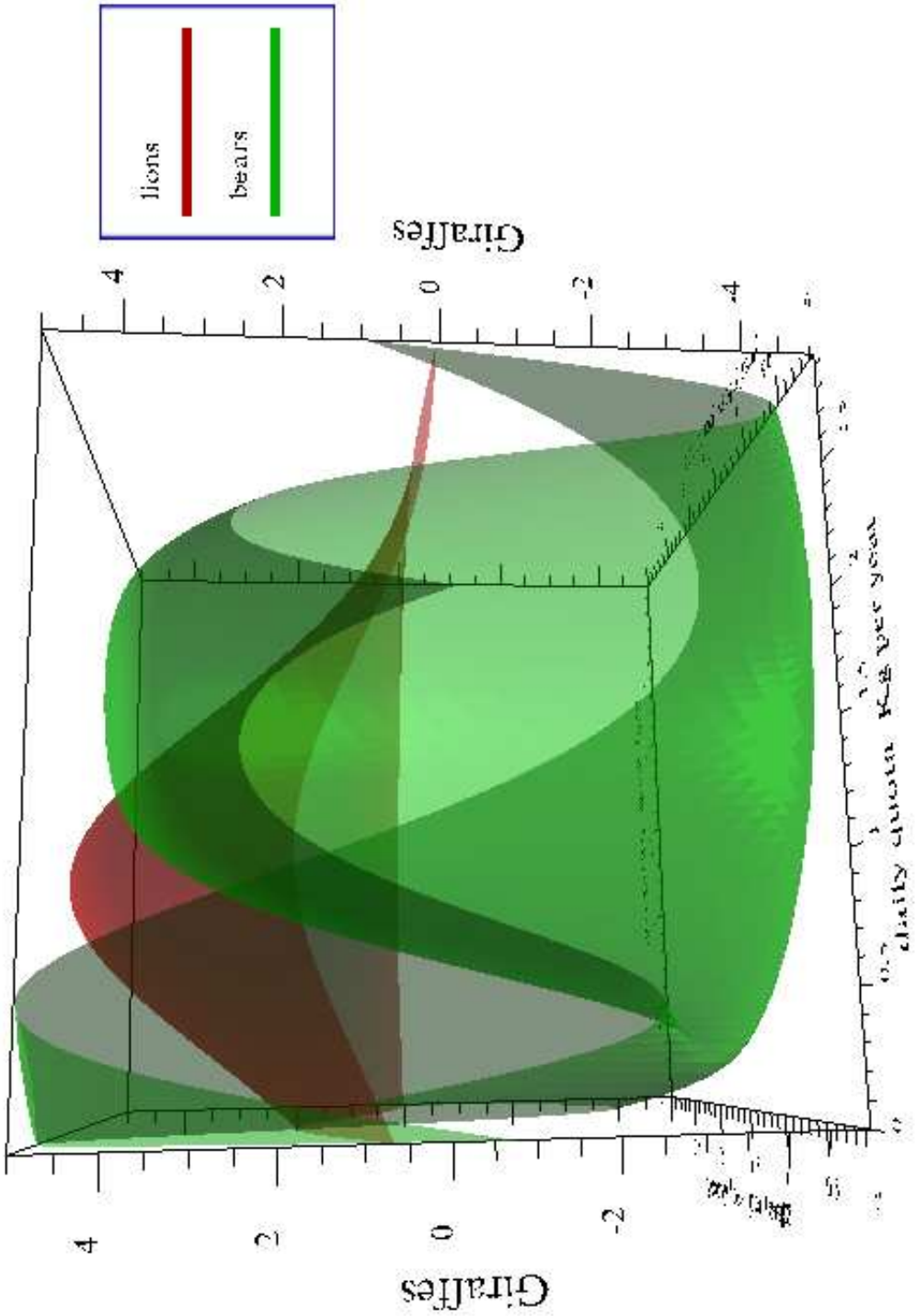

Figure 6: He see two separate functions of two variables as visualised through $PlotSurf$. Transparency can be turned on and off, the light position and colour can be changed, and the material colour can also be changed.

Since the isosurfaces are not necessarily convex, and the data is arbitrary, there is no way of determining the surface orientation when a simple search 'makes contact' with part of the isosurface and grows the surfaces from this first 'contact'. We go into a little detail on how this is done in another article at this site. At any rate, once the visualisation is up on the screen, the user can press 'f' to get the "flipper box" and reverse the surface orientations until the desired effect is obtained.

8 The Future

At the moment, I have stopped working on the PlotFiles, but they shall be revised and added to in future. For instance, *PlotGlobe* is a "place marker" for future visualisations of global data and satellite orbit visualisation.

No doubt I shall occasionally come across mistakes which I shall *have to* correct and do so. Sometimes I might choose to live with a minor error until I am finished working on whatever it happens to be at the time.

One thing I would like to at some point is put all this stuff into a GIT repository. No, not a retirement home for curmudgeonly officers, but Global Information Tracker. This seems a little like overkill for an individual writing stand alone code, but it is still something I would like to learn to do. It would mean that any user can track any changes easily.

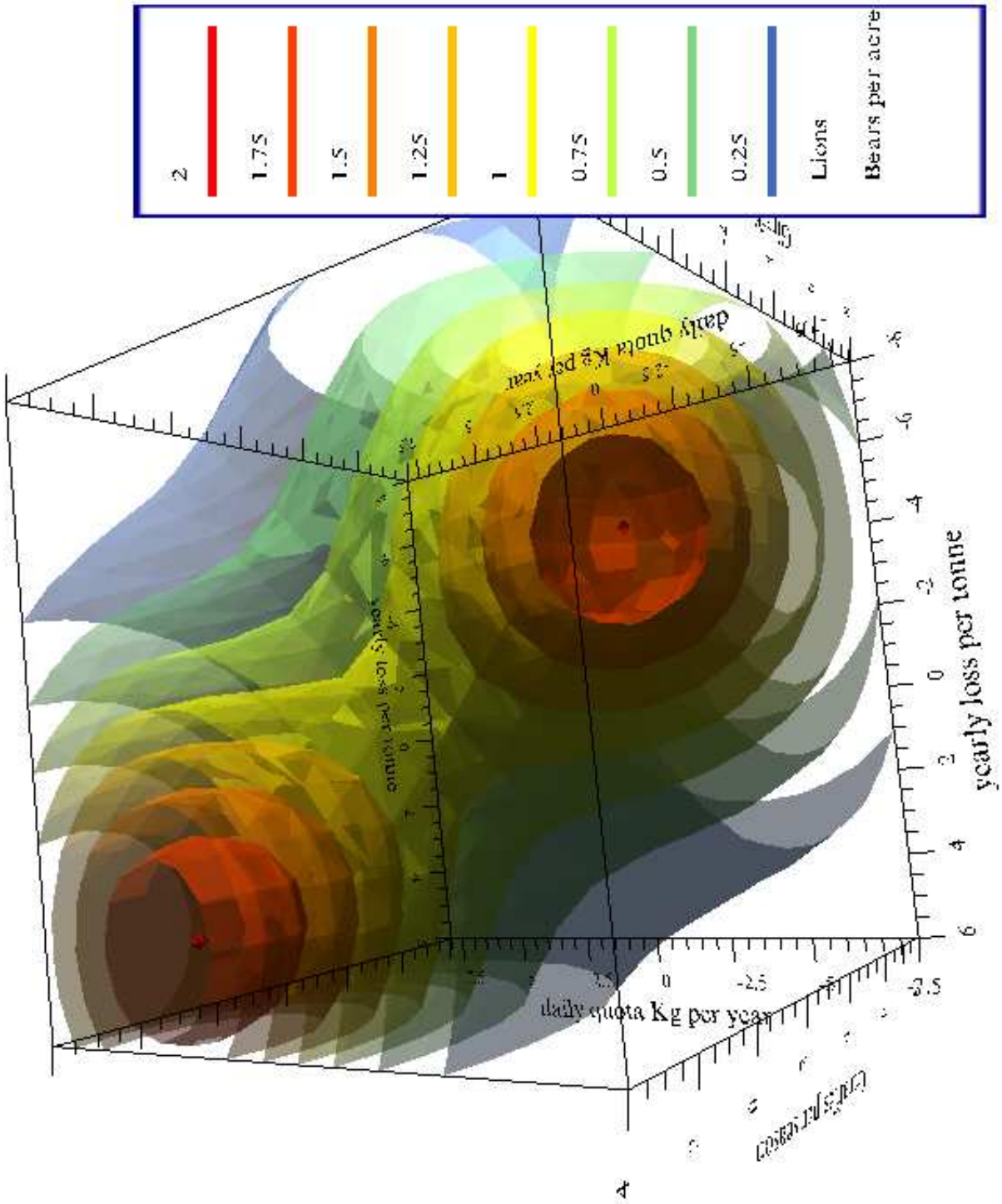

Figure 7: He see isosurfaces of a single valued function of the form $W = W(x, y, z)$ visualised through $PlotVol$. Transparency can be turned off, but it useful to see through to inner isosurfaces that would otherwise be obscured.