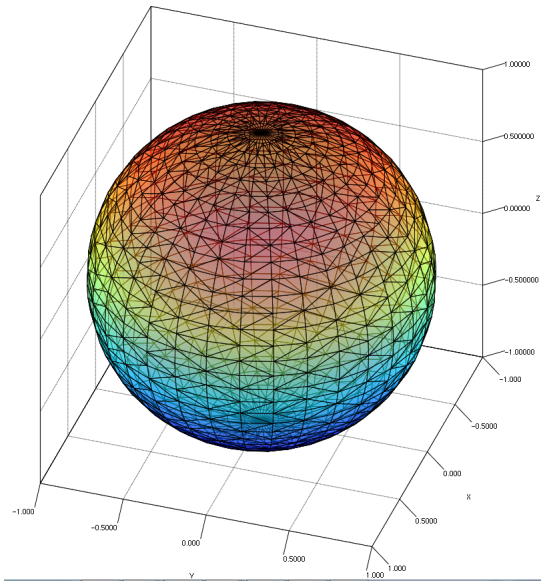
KING'S College LONDON




# OpenGL-based 3D Java graphics in MATLAB using jzy3d - a demo

*Project Waterloo Scientific Graphics Package*

Malcolm Lidierth
Wolfson Centre for Age-Related Diseases

`http://sigtool.sourceforge.net/`

Revised: 9th February 2012

Acknowlegements:

MATLAB code in this document was styled using Florian Knorn's
M-code LaTeXPackage.
http://www.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package

The document was prepared using Pascal Brachet's TeXMaker
http://www.xm1math.net/texmaker/

# Contents

## About this demo

This small package offers a few easy-to-use factory methods for constructing 3D surface and mesh plots in MATLAB using jzy3d. It is for demonstration purposes only: programmers who find this useful, would do best to call jzy3d directly.

jzy3d is an open-source Java package for 3D graphics. For full details see the jzy3d website at `http://www.jzy3d.org/`.

jzy3d uses the Java Bindings for OpenGL (JOGL `http://jogamp.org/jogl/www/`) which allow calls to the OpenGL API (`http://www.opengl.org/`) form Java - and therefore from MATLAB.

This code was developed as a preliminary test for using jzy3d in a Java2D-based graphics package currently being developed as part of "Project Waterloo". The code is a work-in-progress and will change in the future.

## Loading the demo jar file

**Step 1**
To run the demo code you need to put "waterloo-jzy3d-demo.jar" file on your MATLAB Java class path. At the MATLAB command line type:

```
javaaddpath('...\dist\waterloo-jzy3d-demo.jar');
```

where "..." is the path to the dist folder.
The jar files in the lib sub-folder will be added to your MATLAB Java class path automatically and as required. Note that other dependent jar files are included in the standard MATLAB distribution and will already be available on the static Java class path.
**Step 2**
Next, import the required Java functions using

```
import kcl.waterloo.graphics3D.jzy3d.Factory.*
```

## Plotting meshes and surfaces

The Factory package provides static methods to construct a surface or mesh plot. These are

1. createSurface
2. createMesh

Surface and mesh plots are easily interchanged using the *setFaceDisplayed* and *setWireframeDisplayed* methods as described below.

**Examples:**

Create some data using

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
```

## Plotting meshes and surfaces on an orthonormal grid

To plot $Z$ on an orthonormal grid call

```
gr=createSurface(Z);
```

or

```
gr=createMesh(Z);
```

Next, we need simply to add the graphics to a MATLAB container.

```
% Create a figure
figure();
% Add the surface to the figure
[comp, container] = javacomponent(gr);
% Fill the figure
set(container,'Units', 'normalized','Position', ...
    [0 0 1 1]);
```

The result for a surface plot is shown in Fig. 1.

The plot in Fig. 1 could have been a mesh plot if *createMesh* had been used instead of *createSurface*. To convert to a mesh simply call

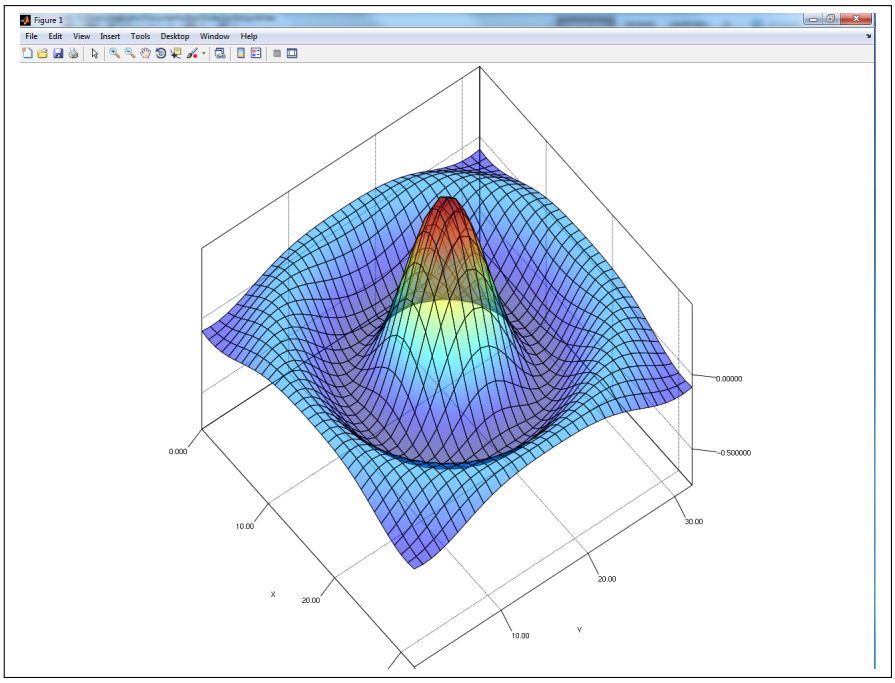



*Figure 1: A surface plot of the data above. The plot can be rotated using the left mouse-button (click-and-hold), moved on the z-axis with the right mouse button and scaled on the z-axis with the mouse wheel. Double clicking causes the plot to rotate continuously.*

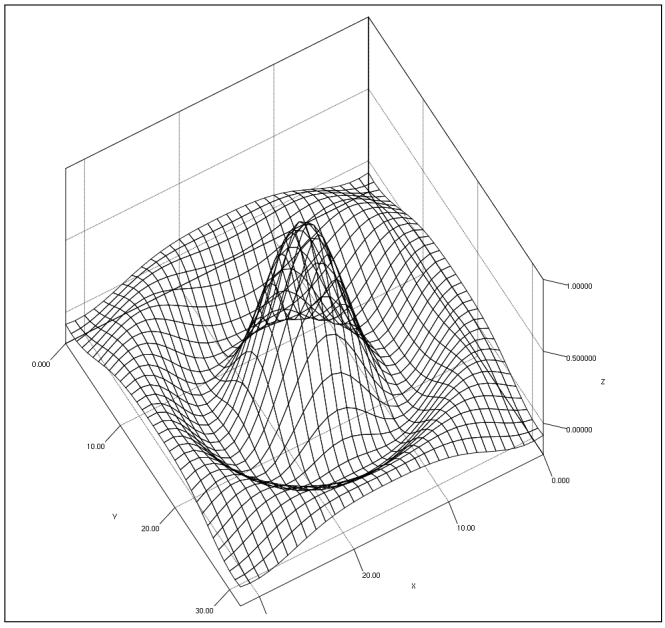



*Figure 2: The plot of Fig. 1 with setFaceDisplayed set false.*

```
gr.setFaceDisplayed(false);
```

Display of the mesh can similarly be controlled with setWireMeshDisplayed e.g.:

```
gr.setFaceDisplayed(true);
gr.setWireframeDisplayed(false);
```

To control the range of the x and y axes, specify x and y as vectors on input e.g.

```
gr=createSurface(-8:0.5:8,-8:0.5:8,Z);
figure();
[comp, container] = javacomponent(gr);
set(container,'Units', 'normalized','Position',  ...
    [0 0 1 1]);
```

produces the same plot as Fig. 1 etc but with the x and y-axes scale -8 to 8. Note that values x and y must *increase evenly*. For an orthonormal grid, duplicate x,y values are not supported.

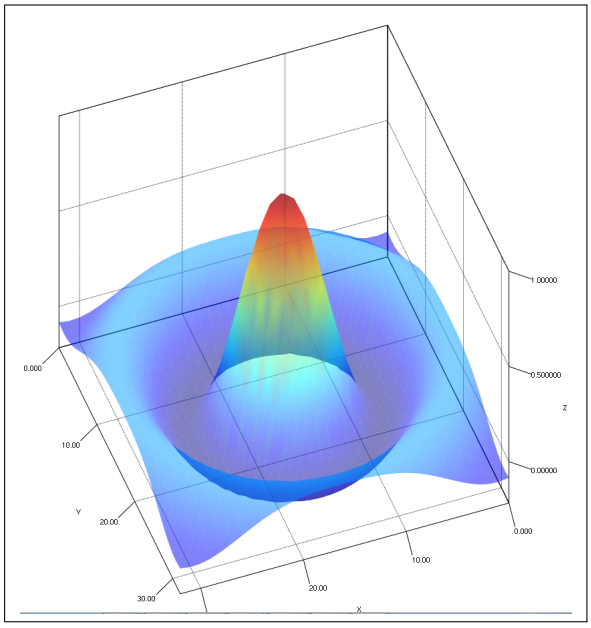



*Figure 3: The plot of Fig. 1 with setWireframeDisplayed set false.*

## Plotting meshes and surfaces using Delauney triangulation

If the inputs for each point are individually specified, *createSurface* and *createMesh* will use jzy3d's Delauney triangulation to plot the data. The data for x,y, and z can be specified either as vectors, or a matrices. Thus for the X,Y, and Z data above,

```
gr=createSurface(X,Y,Z);
>> f = figure;
>> [comp, container] = javacomponent(gr);
>> set(container,'Units', 'normalized','Position', ...
     [0 0 1 1]);
```

will produce the result shown in 4

The Delauney triangulation method is applicable to data that are not regularly spaced. Take a look at the data from the seamount.mat file distributed as standard in MATLAB

```
load seamount
gr=createSurface(x,y,z)
f = figure;
```

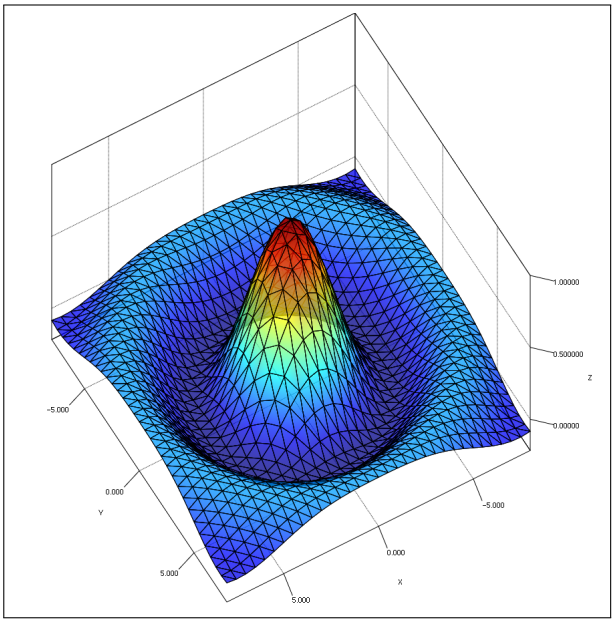



*Figure 4: The plot of X,Y,Z using Delauney triangulation.*

```
[comp, container] = javacomponent(gr);
set(container,'Units', 'normalized','Position', ...
    [0 0 1 1]);
```

gives the plot shown in Fig. 5.

## Limitations

The surface and mesh plotting routines here work differently to the *surf* and *mesh* MATLAB functions. Notably, it is assumed that a single value is supplied for each x,y coordinate pair - i.e. there are no duplicate points. With the following code:

```
[x,y,z]=sphere(32);
gr=createSurface(x,y,z);
f = figure;
[comp, container] = javacomponent(gr);
set(container,'Units', 'normalized','Position', ...
    [0 0 1 1]);
```

as the x,y coordinates are replicated, only one set will be used producing half of a sphere as in Fig. 6.

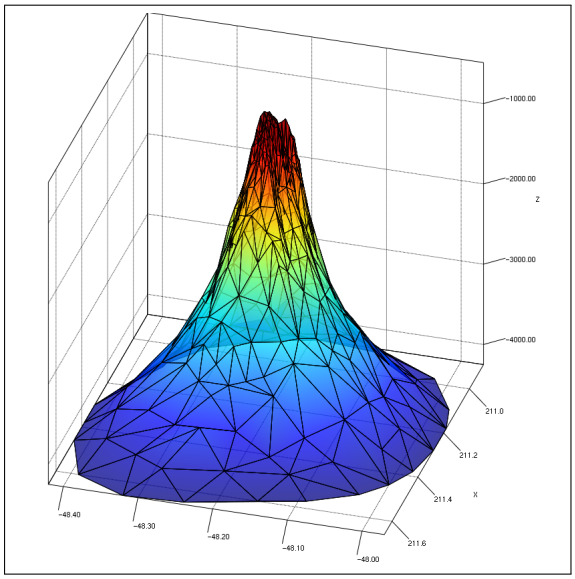



*Figure 5: The seamount.mat data plotted using Delauney triangulation.*

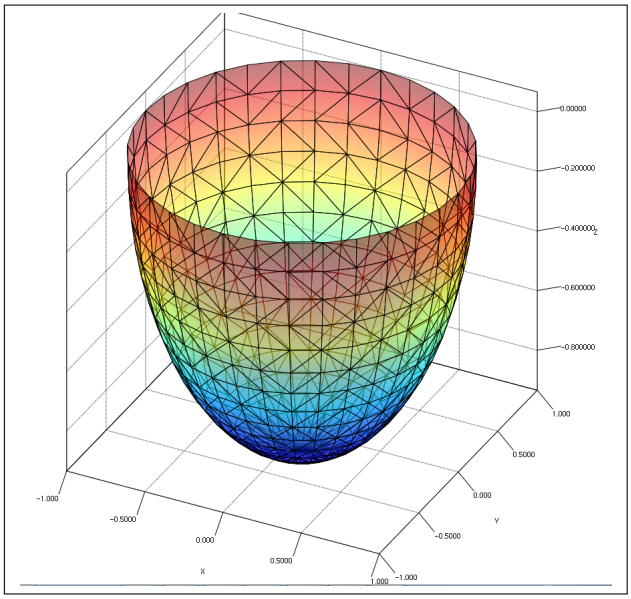



*Figure 6: Half of the required sphere.*

In fact the demo code used here checks the first and last x,y coordinates to see if they match. If they do, it plots the data to produce one half then reverses the coordinate order to plot the other half producing the full sphere shown in Fig. 7.

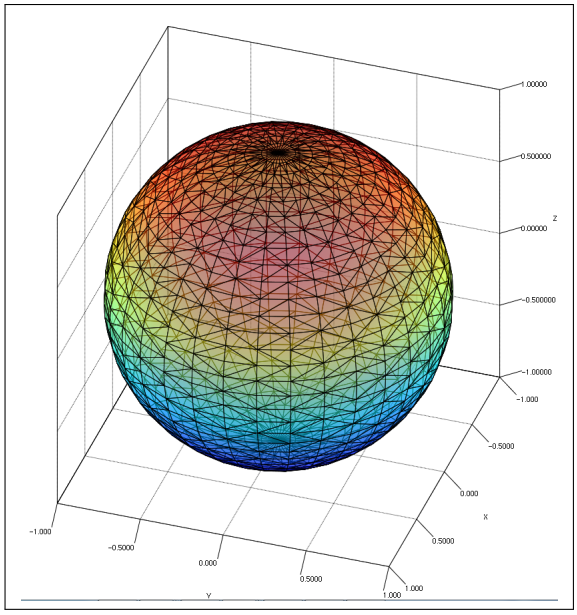



*Figure 7: The required sphere.*

Note that while this works with the output of the MATLAB sphere function with an even input, it is not a general solution. Also matching requires equality of the x and y coordinate's float values - so no tolerance for rounding errors - but this is just a demo after all.