

Paper 342-2010

From Genomes to Galaxies: 3D Graphics in JMP Scripting Language

Kelci Miclaus, Craig Hales, David Barbour, Jong-Seok Lee,
Stan Martin, Russ Wolfinger
SAS Institute, Cary, NC

ABSTRACT

JMP Scripting Language (JSL) offers an interface to OpenGL for three-dimensional graphics. We present several new examples illustrating the aesthetics and interactivity of JSL 3D graphics: a DNA helix, human and bacterial chromosome maps, the Stanford Bunny test model, 3D pie charts, and a bright star and constellation viewer. We highlight the key features of each script, including ways to utilize mouse actions (both movement and click events) and associative arrays. All scripts will be made available online at the JMP Community File Exchange, found at <http://www.jmp.com/community/>.

INTRODUCTION

JMP Statistical software from SAS provides a wealth of visual and statistical tools that are available to you through a graphical user interface. JMP Scripting Language (JSL) allows you to go further to create and manipulate data tables, matrices, and platform reports. Through JSL, custom analyses and graphics can be built to fit your needs, including custom 3D graphics made possible through JSL commands that are derived from OpenGL (which will be referred to as JSL Scene3D commands in this paper). OpenGL (Open Graphics Library) was developed by Silicon Graphics Inc. in 1992, and is a set of commands that provide a software interface to graphics hardware [1] which allows you to create three-dimensional graphics and applications. OpenGL is utilized in data visualization, virtual reality, flight simulation, and most notably, video game development.

The premise of OpenGL is to use geometric primitives such as points, lines, triangles, and polygons to build a 3D object. Commands such as **Translate ()** and **Rotate ()** change the position and orientation of graphics primitives which allows you to build complex shapes. Other commands control the viewing perspective, lighting, coloring, and texture of the surface of the object. OpenGL commands that control how the 3D scene is viewed can most easily be interpreted as the positioning of a camera and lights in a photo shoot. In this paper, the main JSL Scene3D commands that enable a JMP programmer to build a 3D scene in JMP will be discussed and novel examples will be given that highlight different uses of the JSL interface to OpenGL as well as other JSL scripting techniques.

JSL COMMANDS TO CREATE A 3D SCENE

All 3D objects must be placed in a scene in order to be viewed in a new JMP window. This is accomplished with the **Scene Box ()** command. The form should follow that outlined in the JMP scripting guide [2]:

```
myScene=Scene Box(300, 300); //create a 300 by 300 pixel scene box
...(commands to set up the scene)...
New Window ("3-D Scene", myScene); //draw the scene in a window
...(commands that manipulate the scene)
```

A display list containing the commands that will render the 3D object needs to be sent to the scene. The **Scene Display List ()** command is used to create this list, which contains graphics primitives commands or pre-defined commands that create the shapes (such as **Cylinder ()**, **Disk ()**, or **Sphere ()**) that form the whole object. A collection of display lists can be sent to a scene, which is a technique that will be shown in the first example, a rendering of a DNA double helix, and in later examples such as a scene containing multiple 3D pie charts. The scene display list(s) are sent to the scene and executed using the **Call List ()** command. Alternatively, a display list can be sent to the scene inside the **ArcBall ()** command, which will allow the 3D object generated by the display list to be rotated and spun interactively by the user based on a mouse click and movement. The **Update** command must be sent to the scene in order to render the graphics commands that are in the display list. Any time a display list is modified, the scene must be updated to render the new commands.

Other main commands needed in the script are those that define the camera view as well as the positioning and angle of the view. These are outlined in the list below.

- **Perspective (*angle, near, far*):** This command defines the viewing angle (much like the telephoto camera lens where small angle values will zoom into the drawing and large angle values zoom out), and the position of the near plane and far plane. If objects rendered have a larger width than between the near and far plane or are positioned closer than the near plane or farther than the far plane, they will be cut off.
- **Translate (*x, y, z*):** This command allows you to change the position from which the scene is viewed. For example if a scene shows an image of a one inch diameter sphere centered in the middle of the scene box, adding the command **Translate (1, 1, 1)** would change the scene to show a sphere with a diameter that is larger than one inch and it would be shifted to the upper, right corner of the scene box. The **Look At ()** command is an alternative way to set the viewing of the scene.
- **Rotate (*degrees, xAxis, yAxis, zAxis*):** This command changes the angle from which the camera is viewing the scene. For example, if a 3D stick figure man is standing upright, a 90 degree rotation on the Y-axis would have him lying on his side. Sending a display list in an **ArcBall ()** allows you to change the rotation based on mouse movement. Rotation can be likened to a photographer that shoots a model from the left, then the right, then looking up from the ground, etc.

JMP also includes many commands adopted from OpenGL to control the lighting of a scene, such as the positions of light sources and the color of the light (one can control the color of the light that hits the object, the background light around the object, as well as the color of light as it is reflected off the surface of the object). Other commands also control the material properties of the object's surface in ways similar to how the color of light is manipulated. More details on these commands can be found in the examples that follow and the JMP scripting guide [2].

The following sections outline a variety of examples of 3D JMP scripting that highlight both the use of commands summarized above as well as further JSL Scene3D commands that construct the 3D objects themselves.

DNA DOUBLE HELIX: A LOOK AT OUR BUILDING BLOCKS

Our first example of JSL Scene3D scripting was initially written by Laurie Cuffney, an undergraduate student at North Carolina State University, and uses only shapes made by the pre-defined JSL Scene3D commands **Cylinder (*baseRadius, topRadius, height, slices, stacks*)** and **Sphere (*radius, slices, stacks*)** to render a three-dimensional DNA double helix. The code for creating the object is exemplary for a first look at building a 3D object using JMP's interface to OpenGL. The snippet of code below creates a scene display list for drawing one of the two helices that make up DNA (note the use of the JSL functions **Sin ()** and **Cos ()**, which uses the trigonometric functions to define the path of the helix, as well as the use of **Pi ()** to give the circular center):

```
//Create Display List for building a Helix;
Helix = Scene Display List ();
for(i=startangle,i<=stopangle,i=i+c,
    // coordinates are used to position the cylinders via TRANSLATE();
    x0=.45*cos(i);
    y0=.45*sin(i);
    z0=.4*i;
    //use matrix stacks to draw many cylinders at incremental coordinates
    Helix<<PushMatrix;
    Helix<<Translate(x0,y0,z0); //position for drawing the current cylinder
    Helix<<Color(50/255,205/255,50/255); //green
    Helix<<PushMatrix;
        Helix<<Rotate(90,1,0,0);
        Helix<<Rotate(ra,0,1,0); //rotation for drawing current cylinder
    Helix<<PushMatrix;
        Helix<<Rotate(45,1,0,0);
        Helix<<Cylinder(.1,.1,.005,10,10);
    Helix<<PopMatrix;
    Helix<<PopMatrix;
    Helix<<PopMatrix;
    ra=ra+(180/(pi()/c));
);
```

The code above highlights many key features of JSL Scene3D scripting. It creates a display list named "Helix" that contains the commands to build one of the green helices as shown in Figure 1A by incrementally increasing the position and rotation at which a cylinder is drawn by using the **Translate ()** and **Rotate ()** commands. This positioning is made simple by the use of matrix stacks via the **PushMatrix** and **PopMatrix** JSL Scene3D commands.

The matrix stack commands can be thought of as place holders for sets of Scene3D commands. When a set of commands such as **Translate ()**, **Rotate ()**, and **Cylinder ()** are put into a display list, the next occurrence of those commands would by default build with respect to the first set. Instead, we want to create each cylinder with its coordinates with respect to the identity matrix or the origin of the scene. The **PushMatrix** command is analogous to setting up a marker of the location you are at in the scene. The first **PushMatrix** in the code above sets of a marker at the origin, then **PopMatrix** will return to that marker indicated by **PushMatrix** to allow subsequent commands to be with respect to that location. Such commands are useful when building hierarchical models and are used in all the examples in this paper.

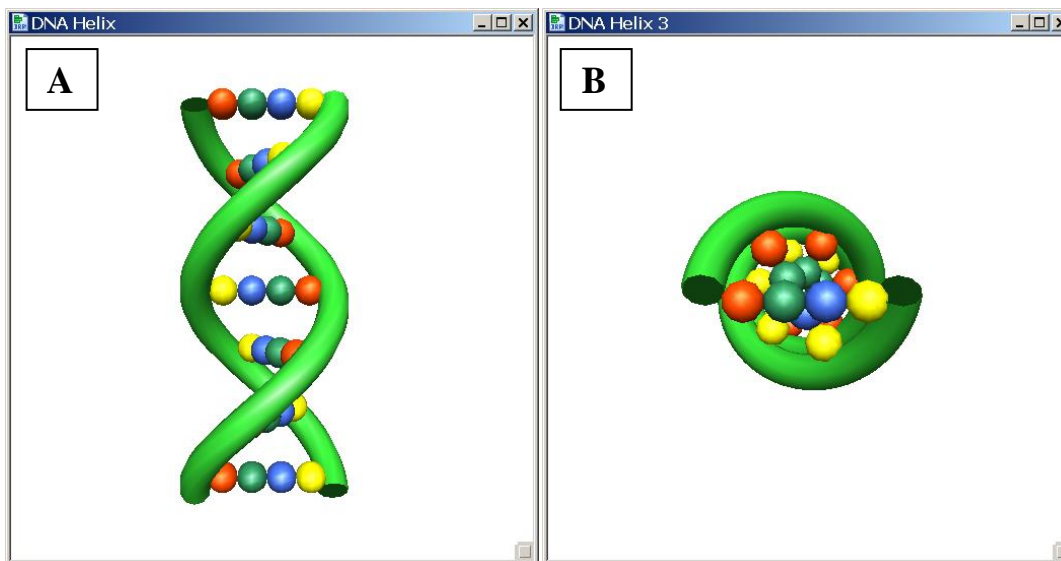


Figure 1: Pre-defined JSL Scene3D commands to create cylinders and spheres were used to construct a 3D graphic of DNA. Figure 1A shows the rendering with the camera angle rotated by 90 degrees on the X-axis. Figure 1B is the view of the object if the `Rotate(-90, 1, 0, 0)` command function is omitted.

The coding example above shows how to construct a scene display list for the helix of our 3D DNA model. The spheres that represent our DNA coding bases (A, T, C, and G) were built in a display list in a similar manner. Below we now show how to render the object in a JMP window by sending these display lists to a scene.

```
//Assemble 3D graphic through commands contained in scene display lists;
DNA = Scene Display List ();
DNA<<Translate(0,0,-1.3);
DNA<<Call List(Helix);
DNA<<Rotate(180,0,0,1);
DNA<<Call List(Helix);
DNA<<Call List(Rows);

//Create a Window containing the scenebox and enable coloring for object surfaces;
scene = Scenebox (500,500);
New Window ("DNA Helix", scene);
scene<<Enable(Color_Material);

//Lighting for the Scene
scene<<Enable(Lighting);
scene<<Enable(Light0);
    scene<<Light(Light0,Position,0,1,2,0);
    scene<<Light(Light0,Diffuse,1,1,1,1);
    scene<<Light(Light0,Specular,.5,.5,.5,1);
    scene<<Light(Light0,Ambient,.1,.1,.1,0);

//Shine of the objects
scene<<Material(Front_and_Back,Specular,1,1,1,1);
scene<<Material(Front_and_Back,Shininess,20);
scene<<Material(Front_and_Back,Ambient_and_Diffuse,0.8,.8,.8,1);
```

```
//Commands to specify viewing space, position the camera view and camera angle;
scene<<Perspective(45,1,10);
scene<<Translate(0,0,-4);
scene<<Rotate(-90,1,0,0);

//Send scene display list in an arcball to allow rotation through mouse actions;
scene<<ArcBall(DNA,1);
scene<<UseHardwareAcceleration(1);
scene<<Update;
```

The code above creates a new scene display list called “DNA” that contains the executed commands from the “Helix” display list and the “Rows” display list (containing commands for the spheres). Note the use of the **DNA<<Rotate(180,0,0,1)** command near the beginning of the code. While the first instance of the **Translate ()** and **Rotate ()** commands in a scene display list will set up how the object is viewed from the camera (as in how we move from the origin of the scene); subsequent calls of the commands after an object has been created in a scene set up how graphics primitives or objects are positioned with respect to each other (in this case how each helix is positioned). This allows you to execute the same “Helix” display list to create the second green helix shown in Figure 1. Examples of setting up the lighting and the material of the object are also shown in the script above. The display list, after positioning to view it from the angle we desire, is sent in an **ArcBall ()** command to allow the user to click down on the mouse and move it to control rotations of the object.

This first example not only showed us a rendering of the building blocks of life, but also the building blocks of JSL Scene3D scripting. Further examples also use and build upon the commands shown in the code samples above.

A CHROMOSOME VIEWER: AN EYE CATCHING ANALYSIS TOOL

This 3D example is an analytical process (AP) in the JMP Genomics software product. JMP Genomics is a software solution that bundles JMP, SAS, and a large set of custom macros for analyzing genomic data. There are millions of locations along the DNA in our chromosomes that show variation from person to person. This variation contributes to the physical differences we see among humans and other species. Similarly, when DNA is translated into RNA, there can be differential expression of our genes. Therefore it is of interest to find such differences that significantly influence onset and progression of diseases such as cancer, diabetes, schizophrenia, and heart disease, as well as other physical traits. A common form of analysis is testing for disease association at a genome-wide level, where genes or portions of DNA are tested for differences between cases and controls at locations that span all the chromosomes. Figure 2 shows a snapshot of output for a graphical display application built using JSL (including Scene3D commands) that allows you to visualize the results of the statistical disease association tests across the human chromosomes. Such a display makes a compelling, attractive summary of an analysis as well as aids in finding regions of the genome that show disease association.

Chromosomes are partitioned into regions called cytobands. The chromosomes in Figure 2 were drawn using the **Cylinder ()** command and the cytobands are colored based on a variable that defines them. P-values are represented by the lines above regions of the chromosomes. These lines were drawn using JSL Scene3D graphics primitives commands which take this scripting form:

```
scene<<Begin(primitive type);
scene<<Vertex(x0,y0,z0)//specify vertex v0
scene<<Vertex(x1,y1,z1)//specify vertex v1
...
scene<<End();
```

where the **Vertex ()** commands defines locations of vertices that will be connected based on the primitive type of graphic chosen. Below is a snippet of the code for drawing a line primitive in the chromosome viewer example.

```
hotspots[i] << Begin( LINES );
...
hotspots[i] << Vertex( c, ybottom, f*diam );
hotspots[i] << Vertex( c, ytop, f*diam );
...
hotspots[i] << End();
```

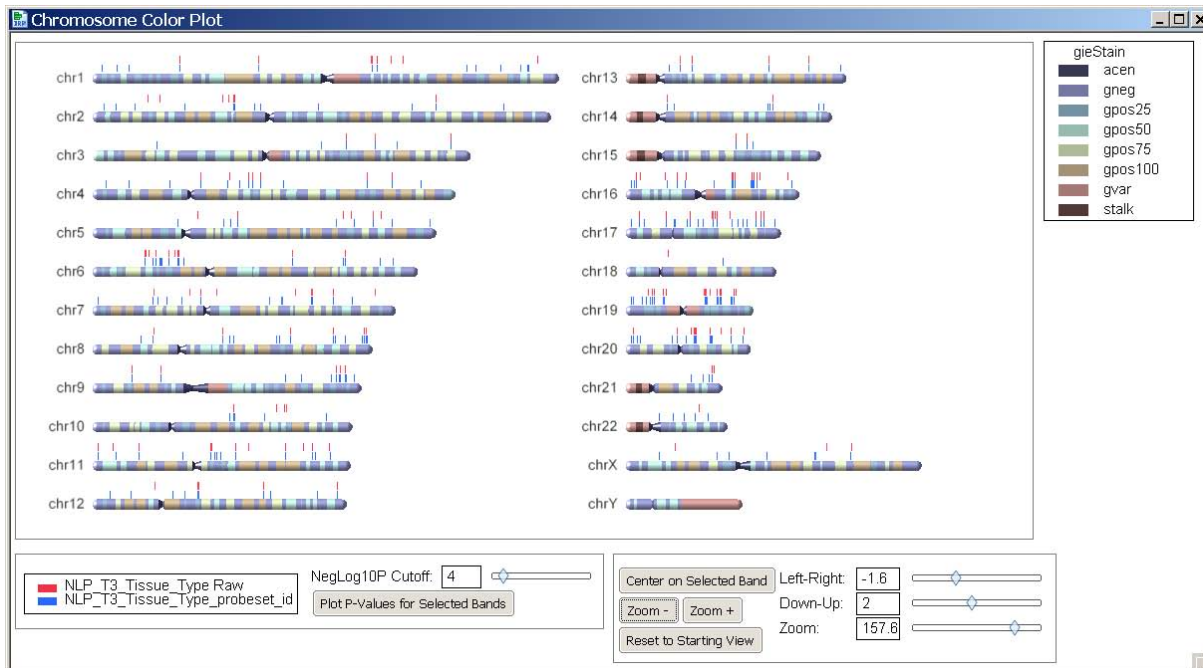


Figure 2: A 3D scene for viewing a set of human chromosomes that can be dynamically changed using JSL controls to send new commands to update the scene. Bands drawn on the chromosomes can be selected and trigger row selection in a corresponding data table.

In this example, many features of the scene have been set up to be controlled by JMP widgets made from JSL functions such as **Button Box ()**, **Text Edit Box ()**, and **Slider Box ()**. These widgets have scripts attached that send new Scene3D commands that change the perspective and the view. In this example, instead of a mouse click influencing the rotation of the objects in the scene, a mouse click can be used to make regions selections. These selections are tied to rows of a JMP data table and further drill down into the statistical results at the selected regions can be explored. Such a mouse action is made possible by defining the **Click3D** function in the following format (more details can be found in the JMP scripting guide [2]):

```
Click3D = Function( {x, y, m, k, hitlist},
    // JSL commands to be executed based on mouse movement/action
    ...
);
```

The parameters **x** and **y** contain the coordinates of the mouse, while **m** shows the state of the mouse (**0** – the left mouse button is up, **1** – the button was just pressed, **2** – the button is down and the mouse is moving, **3** – the button was just released). The parameter **k** has values 1, 2, or 3, respectively representing the state of the *Shift*, *Control*, and *Alt* keyboard key. Responses to all these mouse and keyboard actions can be conditionally scripted to perform within the function above. The **hitlist** contains information about the location of the object the mouse is activating in the scene as well as matrices that name the objects in that region of the scene that have been set up to be selected based on mouse action. When constructing graphics objects in a **Scene Display List ()**, using the **PushName ()** and **PopName ()** commands define objects that can be selected by a mouse action, and work similarly to the **PushMatrix ()** and **PopMatrix ()** commands. The coding example below shows portions of the commands sent to the display list that sets up the cytoband regions to be returned using these commands based on mouse action:

```
// main loop to draw cylinders and name regions that can be returned by a mouse click
For( k = 1, k <= nr[i], k++,
    ...
    chrdisplaylist[i] << PushName( k );
    ...
    chrdisplaylist[i] << PopName;
);
```

Figure 3 shows a zoomed in view (performed by changing the **Perspective ()** command) where a mouse click has

selected region 6p21.1 of chromosome 6. A subsequent *Shift + Click* at another cytoband region on chromosome 6 would select all regions between the two click events. A *Control + Click* would select both the first region and the new region (but not all those between the two).

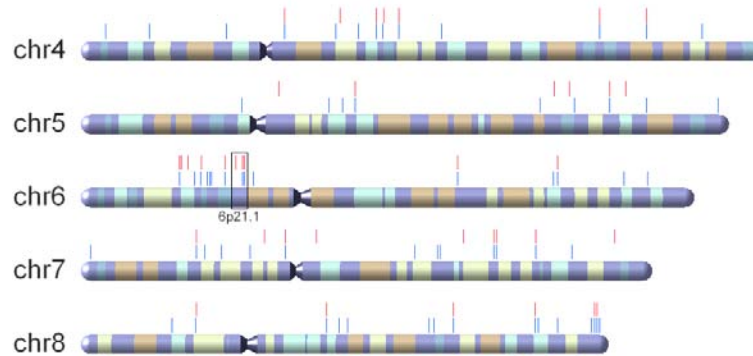


Figure 3: A zoomed view of the chromosome scene showing a selected portion of chromosome 6.

Figure 4 is another screenshot of the output from this JMP Genomics application for a bacterial chromosome, which is a single circular chromosome. This chromosome did not have regions defined by some variable in the data set, so the average p-value across equidistant portions of the chromosome was used for coloring and selection.

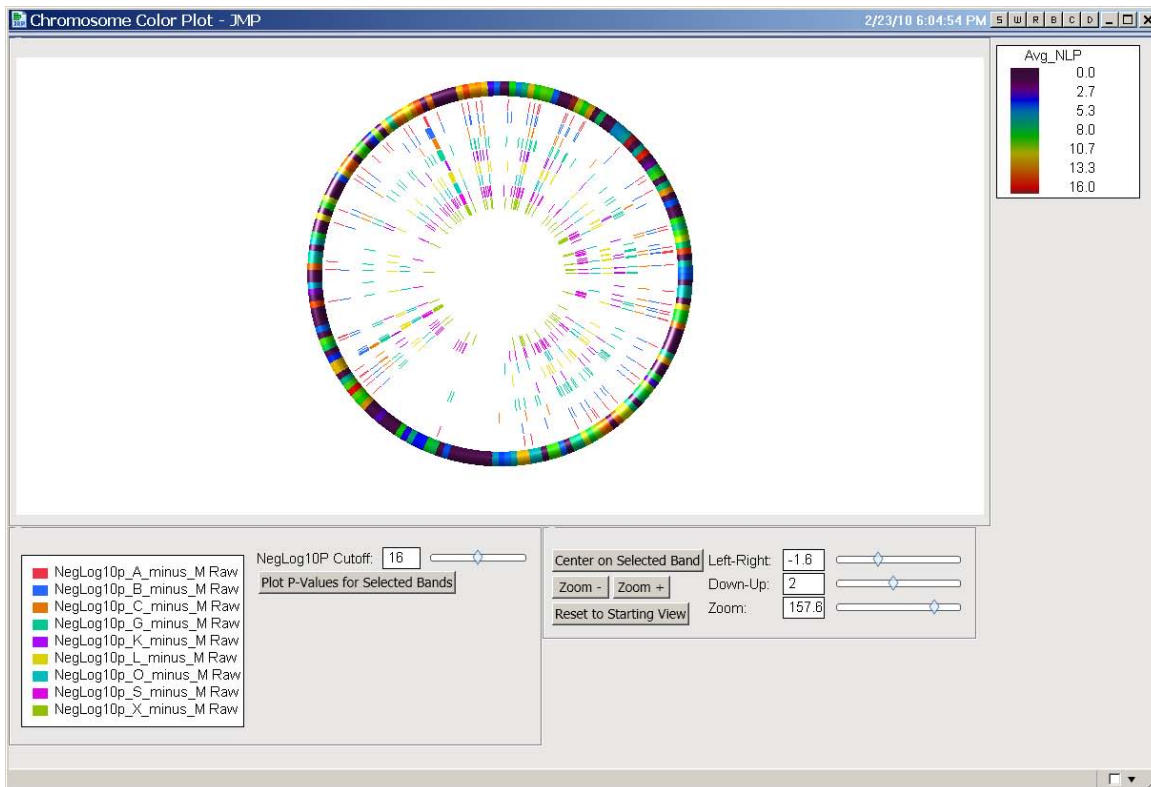


Figure 4: Another genome view displaying a single bacterial chromosome and associated p-values for significance tests performed along regions of the chromosome.

The chromosome viewer example highlights JSL Scene3D mouse click/action functionality as well as shows how JMP widgets can be used to allow the user control over the OpenGL aspects of the window. The following example is similar, and shows how lighting can influence the display of a 3D object as well as how display lists can be programmed to be interactively modified in a scene.

THE JMP MAN GETS A GIRLFRIEND

In the JMP scripting guide, there is an example script for drawing a two-dimensional graph of the “JMP Man”, a graphic that makes up part of icons for JMP and JMP Genomic software. This next example of JSL Scene3D scripting not only creates a 3D display of the JMP man but also cures his loneliness by having him accompanied by a JMP woman. The JMP man now has a girlfriend! This whimsical example contains many of the features of Scene3D scripting outlined previously, including separate display lists for the JMP man and the JMP woman that are sent in separate **ArcBall ()** commands to allow them to be interactively spun. Like the chromosome viewer, the output window contains widgets for modifying the scene. This example shows how the position of the light source sent to the scene can enhance the display.

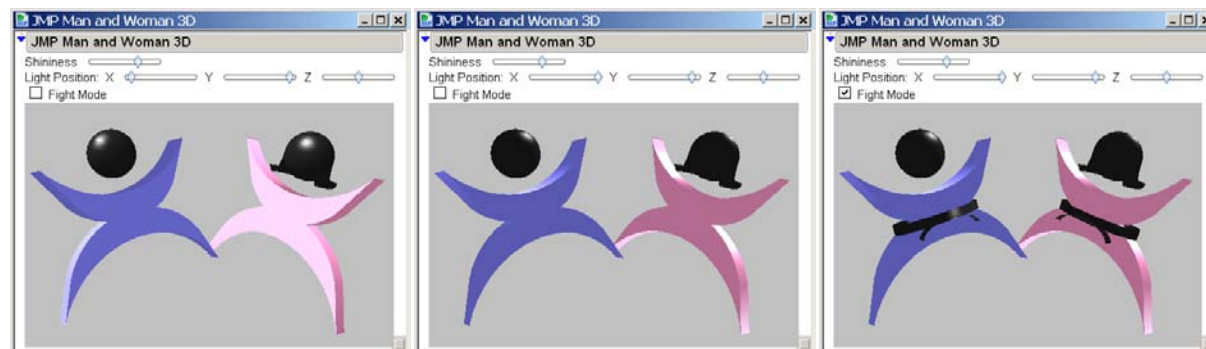


Figure 5: The three scenes above show the rendering of the JMP man and woman. The middle scene shows the change of lighting in comparison to the left-most scene as controlled by the JMP slider bars and the right-most scene is updated with modified display lists (controlled by the “Fight Mode” checkbox) that turn this nice couple into martial artists.

Graphics primitives were used to create the hair on the JMP woman as well as the crescent-shaped bodies (using primitive type *QUAD_STRIP* which draws sets of connected four-sided polygons). In this script a JMP widget controls the position of the light source as well as the shininess of the material (which changes how the light source is reflected from the object). In Figure 5, the differences between the left plot and the middle scene are due to changing the position of the light on the X-axis. Below is a portion of the code that created the window containing the 3D scene.

```
New Window( "JMP Man and Woman 3D",
  ...
  Text Box("Light Position: "),
  Text Box("X "),
  Slider Box(-1,1,light0x,
    Lights << clear;
    Lights << Light(Light0,POSITION,light0x,light0y,light0z,0);
    scene << update;
  ),
  ...
  scene
);
```

In the code above, the **New Window ()** JMP command not only contains the scene but functions for creating the widgets seen in Figure 5. The **Slider Box ()** command in the above code executes a function that changes the X-axis of the light source using the JSL Scene3D command **Light ()**. Notice that the scene display list “Lights” needs to be cleared first so that the new position is not based on where the light source was previously located. The shininess of the material works in a similar fashion by issuing new **Material ()** Scene3D commands. The JMP man and woman example shows how a scene display list can be modified to add new graphics constructs through the “Fight Mode” **Check Box ()** JSL command that adds black belts to the couple (right scene of Figure 5). The function for this check box sets an indicator flag that will create the belt in the display lists and the new display lists are re-executed to show the update. When you set both objects spinning it looks like they are practicing high kicks!

THE STANFORD BUNNY

The Stanford Bunny is a rabbit that may be even more famous than the Easter Bunny; that is, if you are a computer graphics developer. The bunny model was developed at Stanford University and the data were generated based on

a 3D scanning of a ceramic bunny figurine. The model is made up completely of graphics primitives, namely 69,451 triangles. Many other graphics models from 3D scanning, including a dragon made of over 12 million triangles, can be found at <http://graphics.stanford.edu/data/3Dscanrep/>.

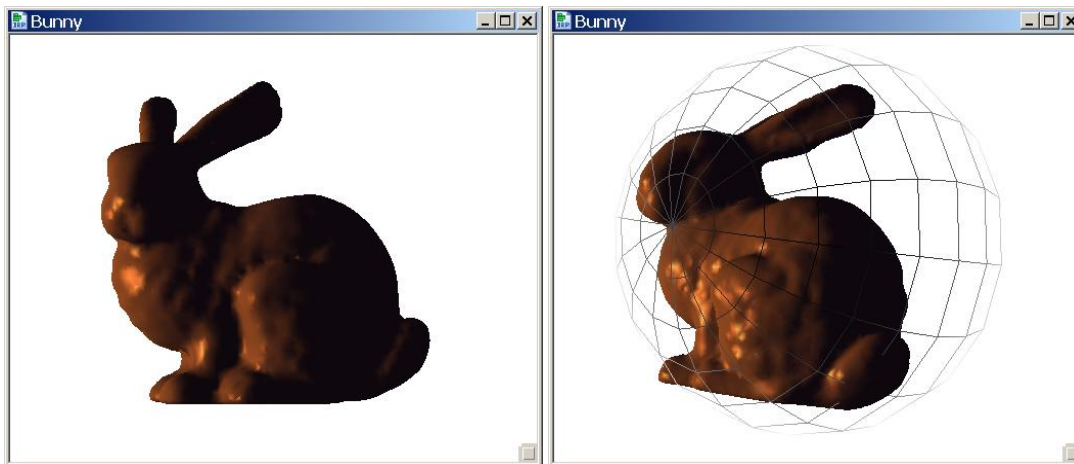


Figure 6: 3D JMP display of a classic computer graphics test model, the Stanford Bunny, made using graphics primitives (nearly 70,000 triangles). The right scene shows the ArcBall to allow rotation by the user.

Figure 6 shows the rendering of the Stanford Bunny in JMP based on normal vector coordinates (for proper lighting of the graphics primitives) and vertex coordinates. The sample of code below shows a loop that iterates through the large matrices of these coordinates to form the bunny.

```
bunny<<Begin (TRIANGLES) ;
for (i=1, i<=ni, i++,
  ii = indices[i,0]+1;
  bunny << Normal (normals[ii[4],1], normals[ii[4],2], normals[ii[4],3]);
  bunny << Vertex (vertices[ii[1],1], vertices[ii[1],2], vertices[ii[1],3]);
  bunny << Normal (normals[ii[5],1], normals[ii[5],2], normals[ii[5],3]);
  bunny << Vertex (vertices[ii[2],1], vertices[ii[2],2], vertices[ii[2],3]);
  bunny << Normal (normals[ii[6],1], normals[ii[6],2], normals[ii[6],3]);
  bunny << Vertex (vertices[ii[3],1], vertices[ii[3],2], vertices[ii[3],3]);
);
bunny<<End ();
```

This model, along with others from 3D scanning repositories, is used to test OpenGL graphics algorithms. Inaccurate projections of the vertices and normal vectors can be quickly recognized with this script. For example, if the normal vectors are not set before the vertices, the lighting of the triangles will cause the model to look much more like a collection of triangles as opposed to representing the tufted fur of the ceramic bunny that was scanned. Figure 7 shows an image, taken from http://www.cc.gatech.edu/projects/large_models/bunny.html, of the triangles that make up the bunny's head.

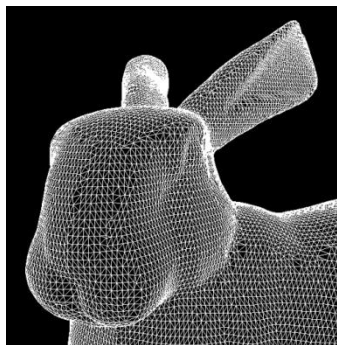


Figure 7: Image showing the thousands of triangle primitives that make up the Stanford Bunny.

THE (IN)FAMOUS 3D PIE CHART

This example is of an updated version of a script that has been available on the JMP File Exchange for some time (<http://www.jmp.com/community/>) that draws 3D pie charts based on your data. The script (like the chromosome viewer plot that displayed p-values) first builds a dialog for entering the variable containing the classes to form the pie slices of the plot. A new feature is the optional "Weight" variable so that the data table can be formatted where each class of the variable is one row with an additional column specifying the percentage of occurrence of that variable class out of the whole. Previously, the data was required to have multiple occurrences of the class in the column specified as "Variable" in order to form the proportions. This is still supported if the "Weight" field in the dialog shown in Figure 8 is left empty. A separate pie chart for each distinct value of a column that is specified as the "Group" variable will be formed (optional).

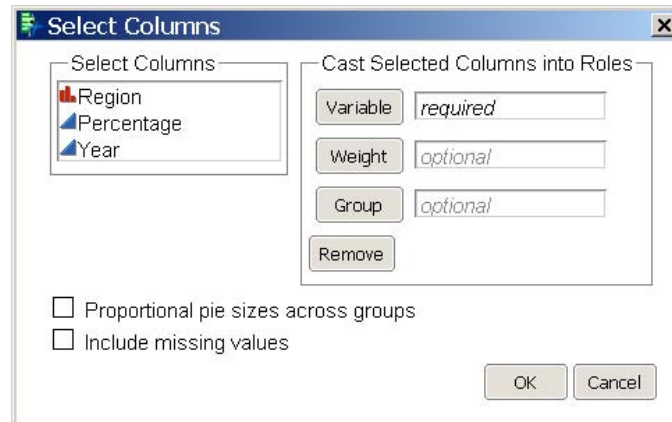


Figure 8: A screenshot of the dialog created by the 3D pie chart script for entering variables for forming the pie charts.

The two scenes in Figure 9 show a 3D pie chart generated by a data table containing the proportions of revenue broken down by region that SAS Institute earned in 2008 (Region as the "Variable", Percentage as the "Weight", and Year as the "Group"); although there was only one value for Year). The scene on the right shows a new feature of the updated script that has a JMP widget labeled "Explosion" that will change the *x* and *y* coordinates of the **Translate ()** command to have the slices of the pie exploding out from the origin. This allows the pieces of the pie to be more easily evaluated and examined.

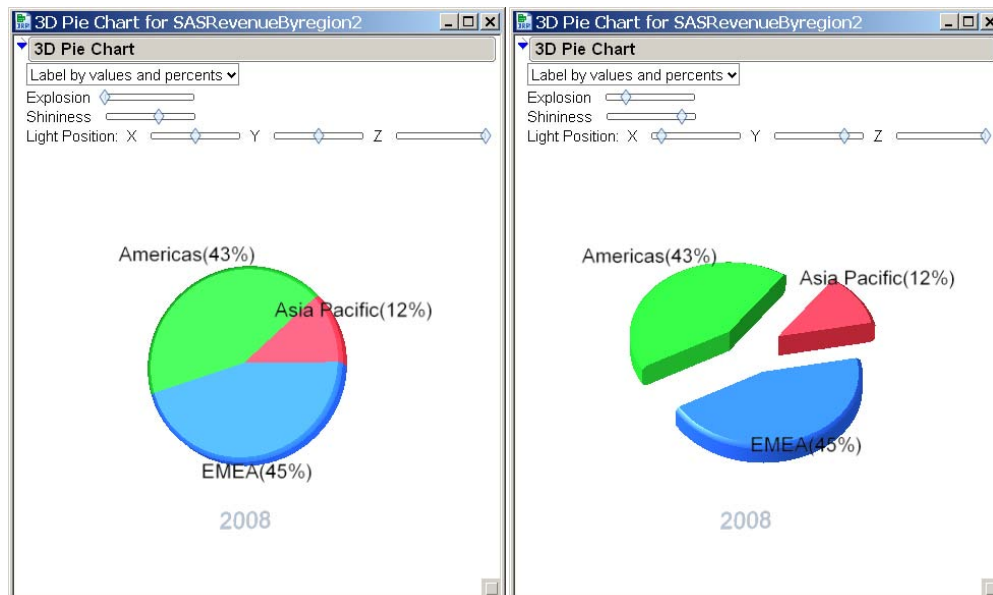


Figure 9: The left scene shows the breakdown of revenue earnings by SAS in 2008 by region. The right scene shows a modification of the pie chart by rotation with a mouse action, changing the lighting position, as well as exploding the pie slices out to more clearly see the components that make up SAS revenue.

One aspect that has not been shown in the previous examples is the use of text in a 3D scene. Text is not a part of the standard OpenGL definition, but the JSL Scene3D command **Text (horz, vert, size, string, <billboard>)** can be used to add text to the scene. In the pie chart script, a JMP **Combo Box ()** gives you a choice of what label for the pie slices are shown ("Label by values", "Label by percents", "No Label" or the chosen option shown in Figure 9 is "Label by values and percents"). The portion of code below shows how the **Text ()** command can be sent to the scene display list. The option **billboard** indicates that the text should rotate in response to a rotation of the object so that the text always faces the viewer.

```

...
if(labelOption==" | labelOption=="Label by values",
    labels[jj] << Text(center, baseline, h/2, char(allDistInLst[i]), billboard),
labelOption=="Label by percents",
    pie labels[jj] << Text(center, baseline, h/2, char(round(value[i]))||"%",
        billboard)
...

```

Two options made available to you through the 3D pie chart dialog, shown in Figure 8, are to have proportional pie sizes across groups and missing value handling. The default setting is to show equal sized pies of a pre-defined maximum size and to not include missing values. If the option for proportional sizes across groups is checked, proportional sized pies will be made based on the weights or counts of classes belonging to each pie group. Depending on what type of missing data is present in the data table (missing values in the Variable column, in the Group column, or both), checking the option to include missing values can result in pie charts that have a slice for "Missing" and/or a pie chart of group "Missing". In the resulting scene, each pie piece can be selected and will link to the data table via the Click3D function. The following programmed options from Click3D are available.

- Selecting a single slice can be done by a left-mouse click + *Shift* key.
- When you have a selected slice or multiple selected slices, you can add selections by a left-mouse click and *Shift + Ctrl* key combination.
- A left-mouse click with *Shift* or *Shift + Ctrl* outside of the pie(s) will clear all previous selection(s).

There are heated debates over the 3D pie chart among the graphics experts and those who desire visually appealing displays. Many experts would insist that you keep your visual output simple and that 3D pie charts can be visually misleading due to volumetric distortion. The use of color and three-dimensional output is considered by many a hindrance to using the plot to understand data; yet 3D pie charts are rampant in presentations and reports and can be fitting when your data represent portions of a whole. Thus, there must be a balance in producing accurate graphics that also satisfy the consumer's desire for a visually appealing display. Figure 10 shows the comparison of a standard bar chart to a 3D pie chart representing variance components data from a microarray experiment.

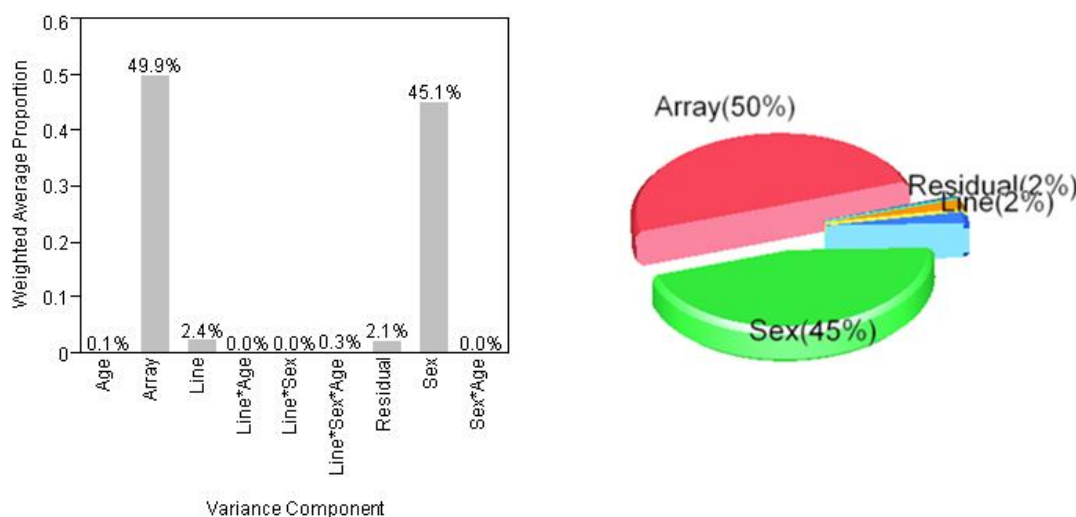


Figure 10: Comparison of a grey-scale bar chart with a 3D pie chart to display variance components. The purpose of the plot is to be able to identify components that explain large sources of variability out of the total variability.

The purpose of this paper is not to promote the 3D pie chart over other graphical displays, but more to promote the possibilities of graphical displays that can be made through the use of JSL Scene3D scripting. For more interesting reading about the debate of the 3D pie chart, see the recent blog post by Russ Wolfinger found at <http://blogs.sas.com/jmp/index.php?/archives/212-I-Like-3-D-Pie-Charts.html>. The images in Figure 10 were taken from this blog.

JMP LOOKS TO THE STARS

The final script we will outline is an advanced example that incorporates all of the JSL Scene3D features highlighted in the previous examples and more. Our little project began based on a text file called the Bright Star Catalogue (BSC) by Dorrit Hoffleit and Wayne H. Warren Jr. and downloaded from <http://cdsarc.u-strasbg.fr/viz-bin/Cat?V/50>. The BSC contains information about the name, position, size (diameter), spectra (brightness), color, constellation membership, and more of over 9,000 of the brightest stars in the night sky (as visible from Earth). The information was imported into JMP and a scene display list was created that uses trigonometric functions and graphic primitives of type *POINTS* to build a spherical 3D bright star viewer of the stars. Star diameter and brightness determines the point size and brightness in the viewer.

The display was difficult to explore and interpret (much like looking up at the stars at night), so further information was added to the scene through additional display lists. Information about the constellation boundaries, directions for connecting stars to form the stick figures of each constellation, and orbital movements of the sun, moon, and visible planets were incorporated to make the "Star JMPer" an interactive exploratory tool for viewing stars and constellations in the sky. Figure 11 shows a screenshot of the application, centered on the constellation Ursa Major (commonly referred to as the Big Dipper). When you look up at the stars and find the big dipper, you may not realize that the Lion representing the Leo zodiac sign, the Twins of Gemini, and the dragon of the Draco constellation lie around it.

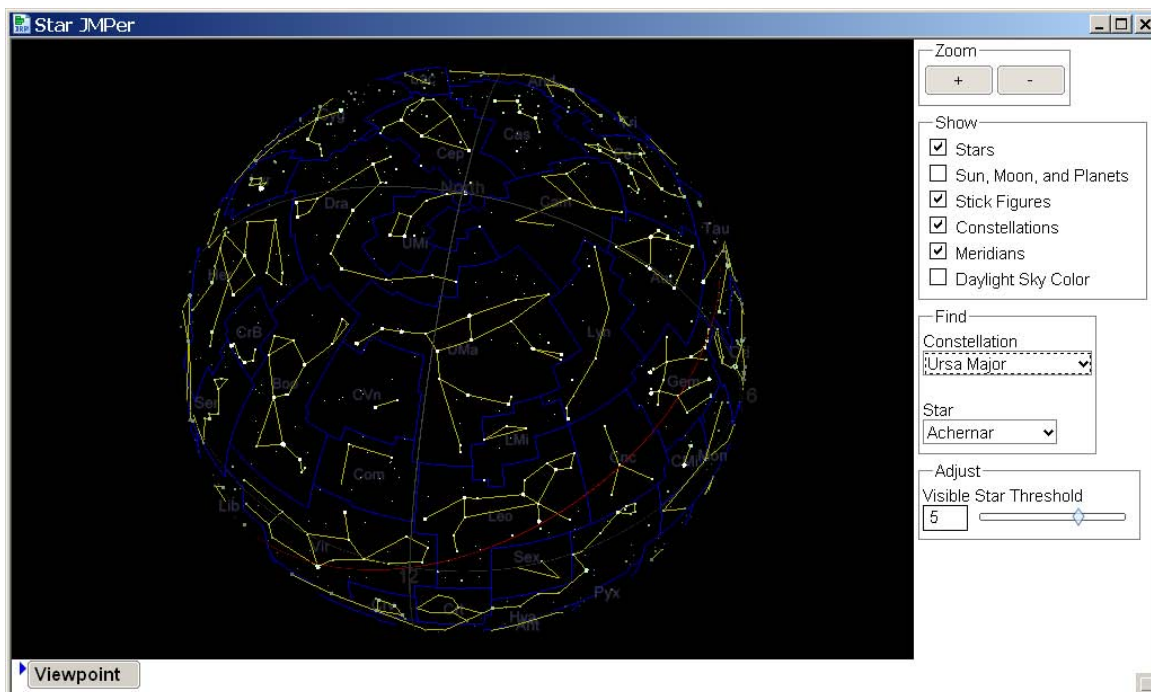


Figure 11: A constellation and bright stars viewer built using JSL Scene3D commands as well as more advanced JSL coding. The view can be changed by mouse action, or by selecting constellations or stars that relate back to positional coordinates through associative arrays.

In Figure 11, it is difficult to grasp the shape of the scene in a static picture. It should be noted that while the display looks to be a convex sphere, the view and movement is actually based from inside the sphere. This is a concave view much like the night sky when you look up at it. If you click and hold in the figure and move the mouse (possible


```

Track2d = Function( {x, y, m, k},
  hitlist2 = stage << Pick( x, y, 5, 5, 1000, 1 );
  ...
  q = Name || " RA=" || char(round(rav[i],2)) || " DE=" || char(round(dev[i],2));
  ...
  label << clear;
  label << PushMatrix;
  label << Ortho2d( 0, boxwide, 0, boxhigh );
  label << Translate( x, boxhigh - y, 0.5 );
  label << Color( 0, 0.7, 0 );
  label << Text( center, bottom, 10, q);
  label << PopMatrix();
  stage << update;
  ...
);

```

The portion of the code above shows the main functionality of the **Track2D** function. The **Pick ()** command sent to the scene (named "stage") returns any star point that falls in the 5x5 pixel box around the current mouse position (x, y). The scene display list named "label" is updated with a **Text ()** command that contains the name of the star and its location that will now show up in the 3D scene as a label when the mouse is still and positioned over a star point.

Lighting of the moon to reflect waxing and waning phases based on the position of the sun required two light sources. The first light source was positioned at the back of the scene for all objects. The moon was made using the **Sphere ()** command and is constantly rotated to always show the front of the sphere while the backside of the sphere is lit from behind by the first light source. The second light source is positioned based on the movements of the sun to reflect its rays as shown in the portion of the code below:

```

// Light0 lights all objects
// position Light0 behind the scene to not greatly alter Light1 effect on the moon
stage << Light(Light0, POSITION, 0, 0, -1.5, 0);
stage << Light(Light0, AMBIENT, 0.5, 0.5, 0.5, 1);

// Light1 orbits the moon and simulates the sun shining on it
// note the moon always faces forward so the effect of Light0 is minimized
stage << Light(Light1, POSITION, xlight, ylight, zlight, 0);
stage << Light(Light1, AMBIENT, 0, 0, 0, 1);
stage << Light(Light1, DIFFUSE, 1, 1, 1, 1);
stage << Light(Light1, SPECULAR, 1, 1, 1, 1);

```

where *xlight*, *ylight*, and *zlight* position the moon's second light source dependent on the locations of the moon and sun in orbit. Figure 13 shows an image of the scene that displays the lighting of the moon.

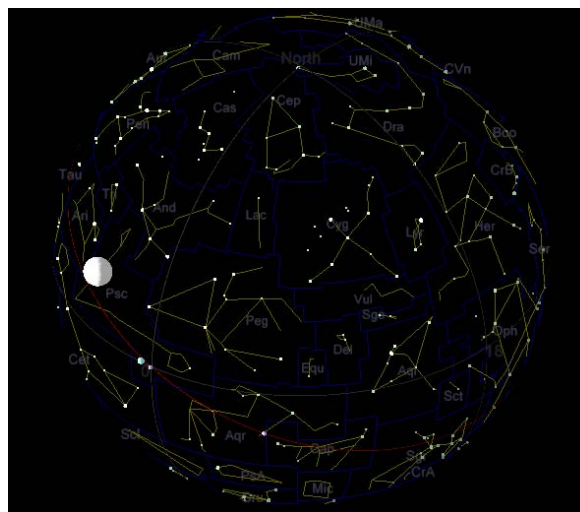


Figure 13: A view of the Star JMPer showing how the positions of the two light sources are used to simulate the lighting of the moon based on the orbit of the sun.

The Star JMPer script combines a lot of complex concepts of JSL Scene3D scripting, but more can be done to enhance this bright star and constellation viewer. The script, which will be made available from <http://www.jmp.com/community/> along with the required data tables, has a section at the beginning of suggestions for additions to the script. These include alternate projection views, adding more locations around the globe for viewing the stars present in the sky at a given time, additional orbital elements such as satellites, etc. If you are interested in applying your OpenGL and JSL scripting knowledge, any enhancements or suggestions are welcome.

CONCLUSION

The adaptation of OpenGL commands into JSL allows you unlimited possibilities for the three-dimensional displays you can create. Through unique and interesting examples, we hope this paper has given you an overview of the important aspects of JSL Scene3D scripting as well as an excitement to create 3D objects in JMP for your own purposes. Along with these scripts, the JMP Sample Scripts directory found in the program files of your JMP installation includes a Scene3D folder containing many more examples of the JMP interface to OpenGL. This directory contains one of the most classic JMP Scene3D examples: the RatMaze.jsl. Many of these shipped examples, along with sample scripts found in the JMP Scripting Guide, are more basic and a good place to start if you are new to JSL Scene3D scripting.

REFERENCES

1. The OpenGL Programming Guide "Redbook" v1.1. http://www.opengl.org/documentation/red_book/
2. JMP® 8 Scripting Guide, Second Edition. 2009, SAS Institute Inc., Cary, NC, USA

ACKNOWLEDGMENTS

The authors would like to acknowledge Laurie Cuffney for her contribution of the DNA double helix script and JMP graphics developer Xan Gregg for his contributions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Kelci Miclaus, PhD
Enterprise: SAS Institute Inc.
Address: 100 SAS Campus Dr.
City, State ZIP: Cary, NC 27513
Work Phone: (919) 531-2843
Fax: (919) 677-4444
E-mail: Kelci.Miclaus@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.