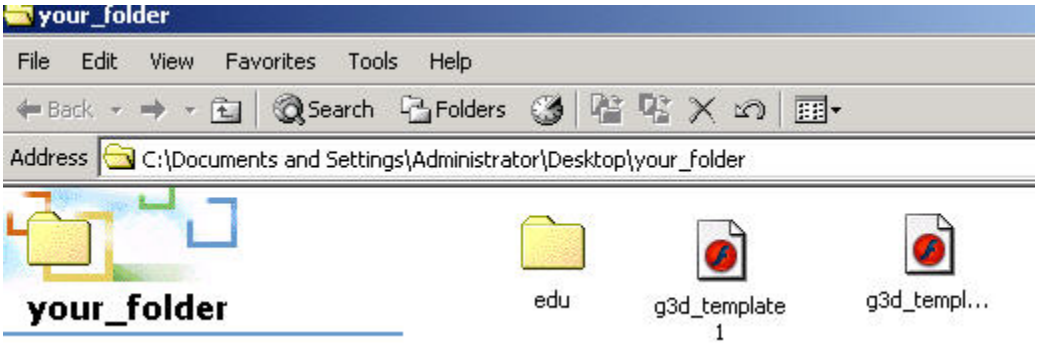
Flash Tools for Developers: 3D Function Grapher
A Guide

This paper is a companion to the online article at the MathDL Digital Classroom Resources "Flash Tools for Developers:3D Function Grapher" by Doug Ensley and Barbara Kaskosz. This paper provides a description of the two 3d function grapher templates and the underlying ActionScript classes presented in the article. The complete source code for the templates and the classes discussed below can be downloaded from the article through the link grapher3d.zip.

Introduction

Download grapher3d.zip file and unzip it in a folder on your computer. You will see a grapher3d folder which contains all the files related to the article. The ones you are particularly interested in are: the folder edu which contains all the necessary ActionScript classes (in a nested sequence of folders), and the two source files for the templates: g3d_template1.fla and g3d_template2.fla. Working right from the folder grapher3d, you can open one of the two template fla files in Flash MX 2004 Professional or Flash 8 Professional and begin exploring and customizing the template. Or, you can copy the edu folder and the templates or one of them into a new folder. For Flash to be able to find the classes, the folder edu must reside in the same folder as the template you are working on:



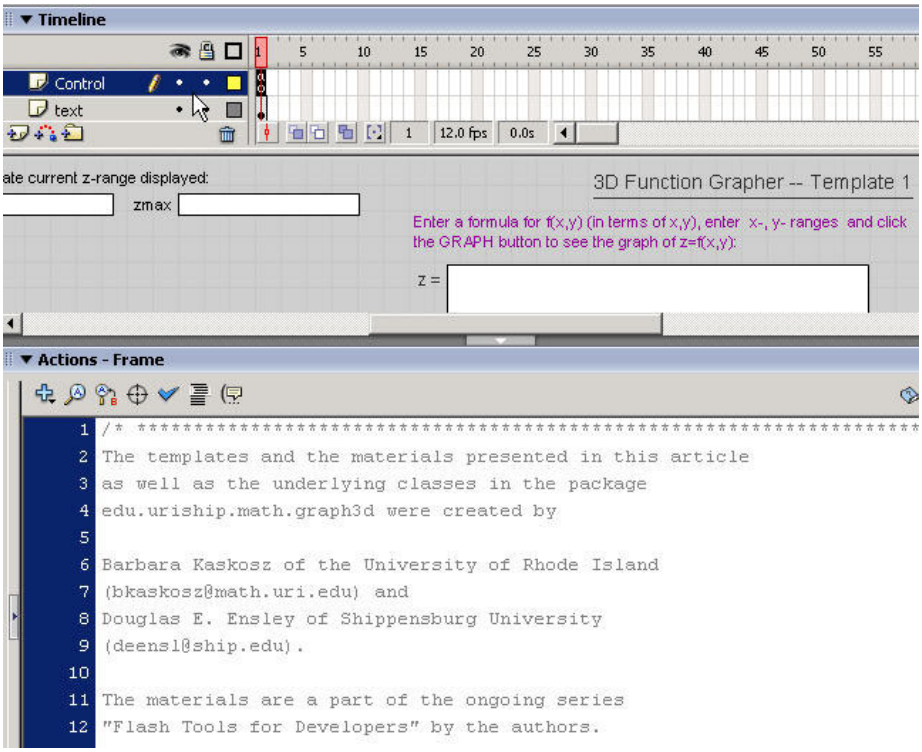
Open your version of Flash, navigate to, and open one of the templates.

Let us begin with g3d_template1.fla. (The screen shots below show Flash MX 2004 but everything looks very similar in Flash 8.) After you open the template, you may want to save it under a new name as your working file, for example, your_grapher.fla. To do that you go to File→Save As, click on Save As, type the new name in the dialog box, and click save. If you are in Flash 8, a dialog box will appear asking you if you want to convert the file (which was originally created in Flash MX 2004) to Flash 8 format. If you plan to continue working in Flash 8, click "Save". If you are in Flash 2004, no dialog box will appear.

In this guide, we are assuming that you are familiar with the elements of the Flash working environment: the Timeline, the Stage, the Properties and the Actions panels, and the Library. If you are not, download fg_guide.pdf from the authors article *Flash Tools for Developers: Function Grapher* and scan through the Getting Started section.

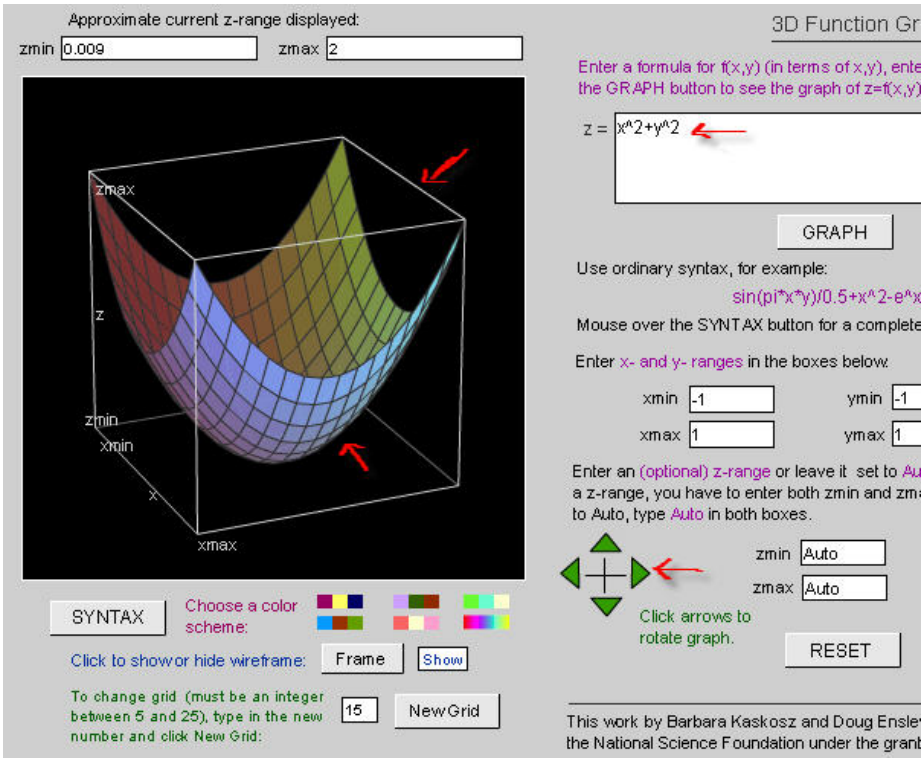
Elements of the Grapher's fla File

When you open g3d_template1.fla, or your working copy of it, you see on the Stage all the elements which were created manually. These are: input and dynamic text boxes, buttons, and static text boxes. The actual graphing board is created programmatically and will appear in the empty space on the left side of the Stage. To see all the code in the file, select the Control layer of the Timeline and open the Actions panel as you see on the picture below. All the code, needed to run the grapher is attached to the first (and only) frame of the Control layer and contained in the classes in the folder edu. Those classes are imported on line 41 of the code in the Actions panel visible below.



Before you do anything else, you should familiarize yourself with how the compiled movie works. To do this, click on the Control item located at the uppermost menu. In the dropdown menu that opens, click on Test Movie:

A new window opens with a compiled swf file playing in the Flash Player:



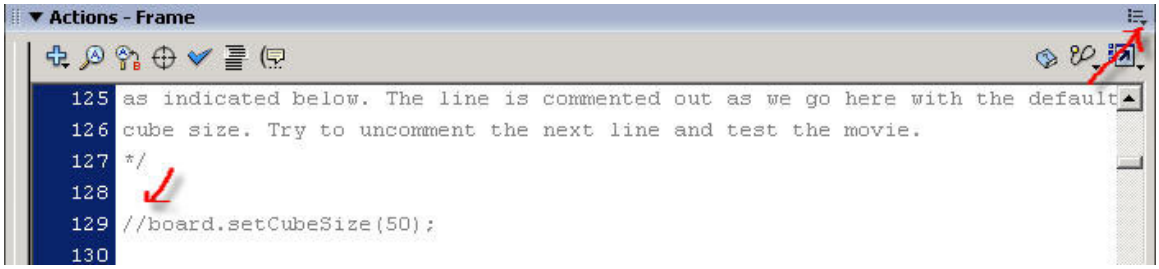
After you see how the grapher works, following the comments in the script in the fla file will become much easier. In the compiled movie, we see the graphing board with the graph of a paraboloid displayed. You see the input box with the formula x^2+y^2 already there and the range boxes with values filled in as well. In the main script, on lines 222-240 and line 315, we preprogrammed the grapher so it would open with the example of a paraboloid. After that, it is the user who enters an xy-formula in the function box and desired values in the range boxes. The graph is then displayed after clicking the GRAPH button.

The graph consists of a surface made up of rectangles and lines. The number of rectangles is determined by the number which we will call a grid or a mesh. The grid is initially set to 15 which means that the square region in the xy-plane, over which the graph appears, is divided into a 15 by 15 grid of 225 squares. The function is evaluated at each of the vertices to obtain the corresponding points, called nodes, in the 3d space. The nodes determine the lines and rectangles in the 3d space. The 3d object is then rotated in the 3d space and the image is projected onto the 2d plane. We shall call the lines that you see on the surface the wireframe. A button visible above gives the user the option of showing or hiding the wireframe. Another button underneath allows for changing the grid. The user can rotate the graph in real time by clicking the green arrows marked on the picture below. Every click causes a rotation about a given axis by a given angle. Each rotation is performed in the 3d space by calculating the new positions of all the nodes. The resulting surface is again projected onto the 2d plane. Note also that right under the graphing board there are buttons allowing the user to change the color scheme of the surface.

The x, y and z axes are displayed in the boxed fashion. The graph appears inside that box. In the script and in this guide, we shall refer to this box as "the cube" and talk about "the cube's size" etc..

Customizing Appearance of the Grapher

Modifying the layout of the grapher, changing sizes and colors of the graphing board and all its elements is very easy and can be accomplished through simple modifications of the script and moving elements on the Stage. For example, scroll to line 129:



and uncomment it by removing the two forward slashes:

```
board.setCubeSize(50);
```

Test the movie now. Everything works the same way but the size of the cube and all the graphs is much smaller. (If the line numbers in the Actions panel do not appear click on the tiny icon in the upper right corner of the Actions panel and check View Line Numbers in the dropdown menu.)

If you want to resize the board itself, go to line 56 where an instance of GraphingBoard3D is created:

```
var board:GraphingBoard3D=new GraphingBoard3D(16,50,330,this,1);
```

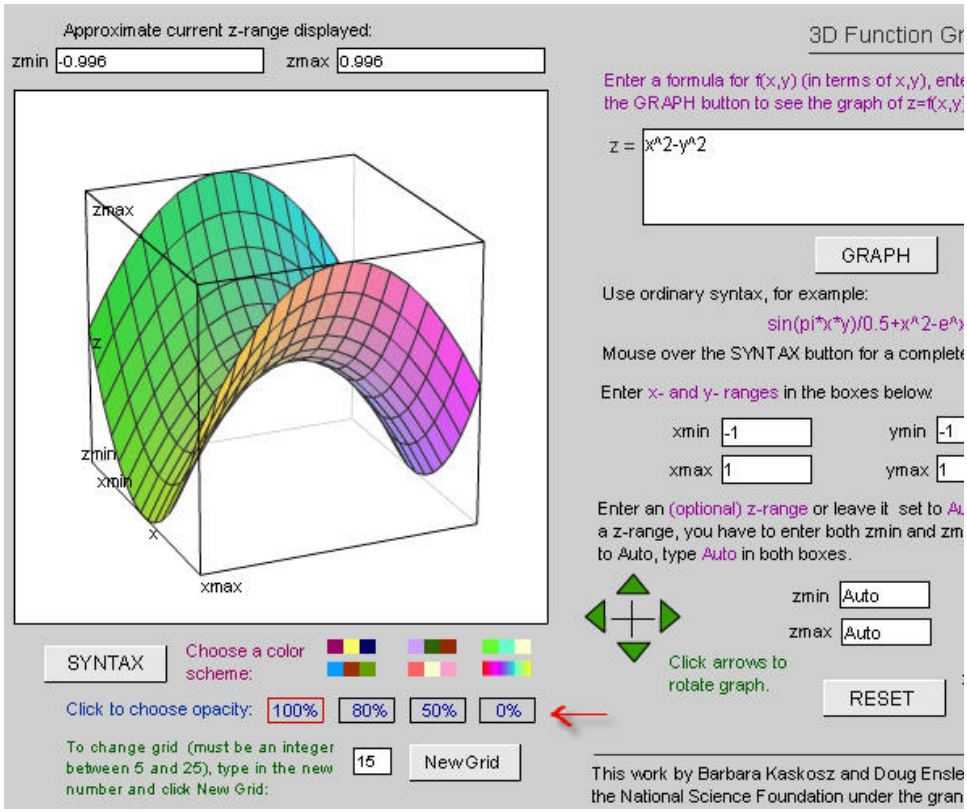
As the detailed comments in the script explain, the third parameter in the constructor determines the size of the square board. Change the line to:

```
var board:GraphingBoard3D=new GraphingBoard3D(16,50,250,this,1);
```

Test the movie. The graphing board is smaller. The cube is fitting in it more or less snugly depending if you commented back line 129 or not. When you instantiate GraphingBoard3D the size of the cube is set to a default value equal approximately to one third of the size of the board. The default value can be changed by the instance method board.setCubeSize(..).You may notice that the message *"Processing"* as well as the error box, which appears when the user makes a syntax error, have adjusted their position and size to fit the new graphing board. Both boxes, one called board.ErrorBox, the other board.WaitBox, belong to and are controlled by the instance "board". To make the rest of the layaout fit a new, smaller graphing board, you will have to manually move the other elements on the Stage.

The file g3d_template1.fla uses all the default values for colors and formats of the grapher's elements. Hence, all the corresponding commands in the script are commented out. To

demonstrate how the colors and other properties of the grapher's elements can be changed through easy modifications of the code, let us close g3d_template1.fla and open g3d_template2.fla. Again, you may want to save a working copy under a different name. And again, test the movie and become familiar with the functionality of the second template:



The second grapher has a different color scheme, different preprogrammed examples. Instead of giving the user the option of toggling the visibility of the wireframe, it gives the user control over opacity of the graph. Play with this feature. The second of the three preprogrammed examples illuminates best the usefulness of opacity in understanding the structure of a surface. Then, close the swf file and get back to the fla file.

The code responsible for the appearance of the "board" is located between the lines 74 and 120. Although the comments in the script explain in detail how to change colors, sizes, and positions of elements, let us look at a couple of examples. Go to lines 74 and 76:

```
board.setBackColor(0XFFFFFF);

board.setBorderColor(0X000000);
```

The commands are responsible for the color of the graphing board and its border. The colors are passed in their hex form. Change them to any colors you wish, for example:

```
board.setBackColor(0XE4E4E4);

board.setBorderColor(0X006600);
```

Test the movie. The background of the graphing board is light silver, the border dark green. Let us change colors of bounding axes and axes labels. Go to lines 82 and 84:

```
board.nAxesFront=0X000000;

board.nAxesBack=0XCCCCCC;
```

Change them to:

```
board.nAxesFront=0X006600;

board.nAxesBack=0X66CC33;
```

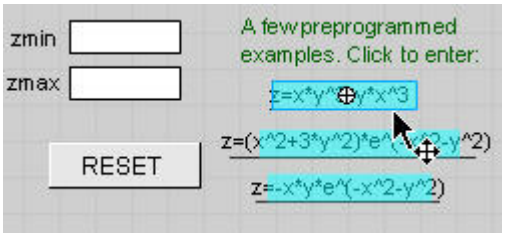
Change line 118 to:

```
board.setLabelsFormat(0X006600,10);
```

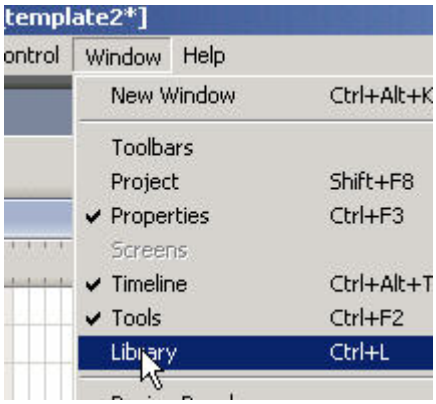
Test the movie. Axes and labels are in green, the font of the labels is smaller. You can adjust colors for the wireframe, the corresponding command is on line 78. It is commented out as the dark gray, 333333, is the default. But you can uncomment the line and change the color.

You can change colors of the border, background, and font of the Error box. The corresponding command is on line 92. You can change its size and location (line 104).
Although the Error box belongs to and is controlled by "board", it does not have to physically be located within the graphing board. You can position it wherever you want.
Similarly, you can adjust font color and size of the Wait box (line 116) as well as text displayed. The position of the Wait box can also be chosen arbitrarily (line 106). The Wait box has no background or border.

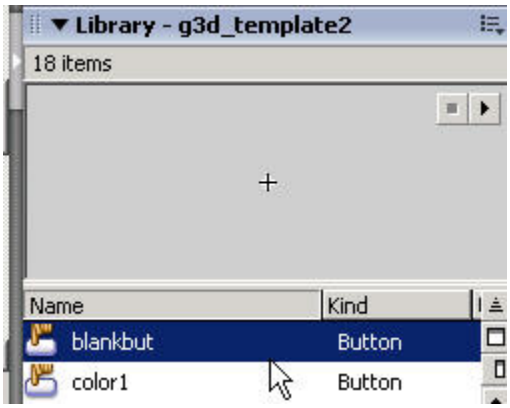
The opacity values given to the user to choose from can be easily changed by obvious changes on lines 799, 826,853, and 880. (Of course, if you change those values you have to manually edit the corresponding buttons so they reflect the right percentage values.) Similarly, you can easily changed preprogrammed examples by editing parts of the code between the lines 938 and 996. Editing the corresponding formulas on the Stage is easy as they are simply static text boxes not buttons. What makes them behave like buttons are the so-called blank buttons placed on top of the text in the fla file:



These buttons are invisible in the swf file. All three of them are instances of a blank button, named blankbut, in the Library. If the Library is not open, go to the Window menu on the top and check Library:



The Library opens in the lower left corner and you can see the blank button there:



When you drag the invisible button from the Library window to the Stage, it becomes visible as turquoise rectangle. You can give it an instance name, in our case butExample1, butExample2, butExample3, and resize it as you wish through the Property panel. Since resizing affects only a given instance, one blank button in the Library allows you to produce any number of buttons of different sizes on the Stage. You can copy the blankbut from one Flash file to another, so once you have one blank button, you don't have to learn the trick for creating it.

We learned to modify the appearance of the grapher. Changes in its functionality require delving deeper into the code.

The Basic Outline of the Program

The three essential functions in the script you see in the Actions panel of each of the two templates are:

-- prepGraph( )
-- drawGraph( )
-- procInput( )

It's easier to look at them in the first template, so let us assume in this section that we have g3d_template1.fla open.

- **prepGraph( )** begins on line 355. This function first clears the board through the board.resetBoard( ) method and resets most of global variables to their initial values. Then, the function collects the user's input from all the input boxes; that is, the user's formula for the function, which is stored in a local variable inpString, and the user's range entries. The range entries have to be numeric, so the function only checks if they indeed are. If not, the function quits and sends an error message to the Error box through board.showError(mes) method. If the range entries are fine, the user's function formula is parsed. Here, the instance of MathParser, procFun, created earlier is used. First, the formula is compiled via procFun.doCompile(inpString) method. If syntax error is detected, the function quits and sends error to the Error box. Otherwise, the multidimensional array, fArray, of nodes (the number depends on the current value of the grid logged in as the property board.nMesh). fArray consists of functional values for the nodes. The vertical coordinate for each node is obtained via the evaluator method of MathParser, procFun.doEval([…,…]). Then, coordinates of each point of fArray is translated into their pixel values resulting in pArray. fArray is modified to only reflect legality of the value at each node. Finally, the local variable pArray is sent to the "board" via board.setPixArray(pArray), and the local variable fArray is sent via board.setFunArray(fArray). At this point, "board" knows the 3d positions of all the nodes and which of the nodes reflect points where the user's function is defined and should be graphed.

  If the range for the dependent variable is set to "Auto", prepGraph( ) function calculates also the range for the dependent variable.

- **drawGraph(M)** function begins on line 534. The parameter M is an array which corresponds to a given rotation matrix. Each rotation matrix passed to the function drawGraph(M) is calculated using the static method of the MatrixUtils class, MatrixUtils.rotMatrix(x,y,z,a). The initial rotation matrix, iniMatrix, for each graph is calculated on lines 288-303. The matrices which result from clicking on the arrow buttons, are calculated in the code corresponding to the arrow buttons, for example, butXup.onRelease function, line 771.

  The function evokes the methods board.drawSurface(M) and board.drawAxes(M). These are two instance methods of the GraphingBoard3D class.

  board.drawSurface(M) rotates all nodes of the function's graph. It also assigns colors to each of the rectangles, colors logged in the property board.aTilesColors. (This array of colors corresponding to the user's choice of a color scheme was previously calculated by a static method ColorUtils.setUpSquareColors(*number*) of the ColorUtils class.) board.drawSurface(M) calculates the distance from the center of each rectangle to the camera and sorts the rectangles in the proper order to subsequently draw their projections from the back to the front. (New ActionScript 2 array sorting methods are used to accomplish this efficiently.)

board.drawAxes(M) rotates the boxed axes and places them on proper depths. The code for the method (contained in GraphingBoard3D class) is very long, but it actually runs faster than more conventional methods involving face culling.

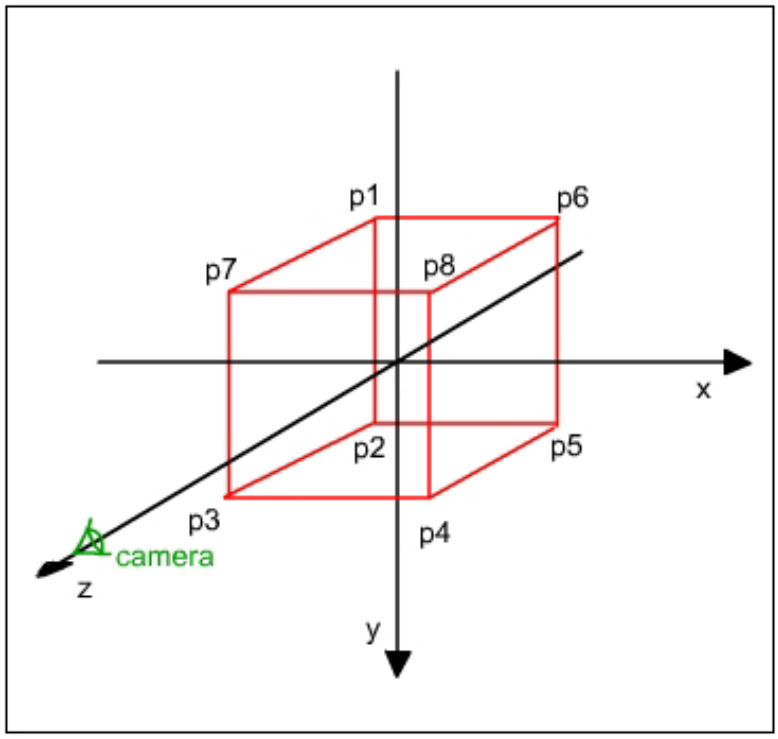
In the last step, drawGraph(M) function rotates the current vertical axes of the bounding cube so the vertical direction remains always vertical.

- **procInput( )** function begins on line 317. First, it calls prepGraph(..). If an error in syntax is found, the function quits. Otherwise, the function calls drawGraph(iniMatrix). The function procInput() is evoked only by the GRAPH button, the RESET button, the New Grid button, and the example buttons. It is also called initially to display the initial paraboloid. procInput( ) function is needed when input for a new function or range must be processed. All other buttons, color buttons, wireframe on/off button, arrow buttons, opacity buttons evoke only drawGraph(M) function.

The three functions discussed above coordinate actions of all the classes in the package edu.uriship.math.grapher3d, and do the initial processing of the user's input. All really essential code is contained in these classes.

The 3D Engine

The 3d mechanism that we use to create graphs, in essence, keeps the camera and the coordinate axes stationary and rotates the cube and a graph of a function. Within the script, we have to use Flash's coordinate system, so what are the x,y, and z axes to the user become z, x, and y axes to Flash and to us when we write or read the program. With the way the movie clip in which all the dynamic drawing is performed is positioned (this is all done by the GraphingBoard3D class), the coordinate system in which we operate looks as follows:



The x and y axes are located on the screen, the z axis is perpendicular to the screen, pointing away from the screen. The stationary camera (that is, our point of view) is located on the z axis at the distance given by the fLength instance property of the GraphingBoard3D class. (The names p1,…,p8 of the vertices are only important if you wish to delve into the code of GraphingBoard3D class.)

The matrix of rotation MatrixUtils.rotMatrix(a,b,c,theta) is calculated for a given axis (a,b,c) and a given angle theta in degrees. The nodes of the bounding cube and the nodes of a given surface are then rotated. They are sorted with respect to the distance of the center of each rectangle to the camera and drawn using the painter's algorithm.

In the earlier version of the grapher, published earlier by one of the authors, each rectangle was a separate movie clip. The distance of each clip from the camera determined there its depth, so any kind of explicit sorting was avoided. That method proved slower than the one presented here.

The Description of Custom Classes

The package edu.uriship.math.grapher3d contains five custom classes: CompiledObject, MathParser, GraphingBoard3D, MatrixUtils, ColorUtils. Here is the description of each class.

- **CompiledObject**

This simple class defines a datatype that is returned by MathParser. The constructor takes no parameters.

```
var compObj:CompiledObject = new CompiledObject();
```

Every instance of CompiledObject has three public properties:

`compObj.PolishArray` -- an array. When compObj is returned by MathParser's doCompile method, the property represents a parsed mathematical expression in the Polish notation. Default value [].

`compObj.errorMes` -- a string. When compObj is returned by MathParser's doCompile method, the property represents a specific syntax error message. Default value "".

`compObj.errorStatus` -- a number 0 or 1. When compObj is returned by doCompile, 0 corresponds to no error found, 1 to error found. Default value 0.

Within the two templates, we only use the instances of the class which are returned by MathParser's doCompile method.

- **MathParser**

This class is the engine behind parsing the user's input. The constructor takes an array of strings as a parameter. For example:

```
var procFun:MathParser = new MathParser(["x","y"]);
```

The array of strings represents names of variables that will be recognized by the instance of MathParser. In the example above as well as in our templates, we use two variables "x" and "y". There can be any number of variables, e.g.: new MathParser(["x","y","z"]), and they can have names longer than one letter. If you do not want your instance of the parser to allow variables, enter the empty array into the constructor: new MathParser([]). Constants "pi" and "e" are automatically recognized and evaluated by the parser; do not enter them into the constructor.

Every instance of MathParser, below we call an instance procFun as in our script, has two public methods:

`procFun.doCompile(`*`string`*`)` -- this method takes a string (typically a string entered by the user) and returns an instance of CompiledObject. To give an example, we repeat below some of the code from Template 1. The code resides within the prepGraph function.

```
var inpString:String=fInputBox.text;

var compObj:CompiledObject;

compObj=procFun.doCompile(inpString);
```

If `compObj.errorStatus==1`, we know that the user has made a syntax error and we can send `comObj.errorMes` to our error box for display. If `compObj.errorStatus==0`, there is no error and we can send

```
compObj.PolishArray
```

to the evaluator method, doEval, of MathParser. The method is discussed next.

`procFun.doEval(`*`array, array`*`)` -- this method takes two arrays as parameters. For the method to do what you want it to do, the first array has to represent a mathematical expression in the Polish notation returned by the doCompile method, the second provides values of the variables recognized by the parser listed in the same order as the order in which the variables were passed to the MathParser constructor. In our templates, there are two variables, "x" and "y", so the second array has two entries, for example, [curz,curx]:

```
procFun.doEval(compObj.PolishArray,[curz,curx]);
```

(curz, curx are numerical variables which were defined earlier in the script. Let's remember that our z axis is the x axis to the user, and our x axis is the y axis to the user. That explains the strange notation.)

The complete list of functions that MathParser recognizes as well as all the syntax rules are described in a movie clip that appears in each of the templates when the user mouses over the SYNTAX button.

It should be noted that MathParser in this package differs slightly from the MathParser class in the packages which came with the author's two previous articles. The earlier parser would do the job just fine. We changed slightly the evaluator method in this package to make it a little faster.

- **GraphingBoard3D**

This class is responsible for creating all visual elements of our graphers except for those that were created at the authoring time. Its constructor:

```
var board:GraphingBoard3D = new GraphingBoard3D(16,50,330,this,1);
```

and its methods are used and commented exhaustively in the templates' scripts. The constructor's parameters are respectively: x and y coordinates in pixels of the upper left corner of the square graphing board relative to the target movie clip, the size in pixels of the square, the target movie clip, the depth in the target movie clip.

Below we list public instance properties and methods. For simplicity, we call a given instance "board" as in the script, and enter examples of possible values and parameters. Normally, "board" would be replaced by the name of the instance you create.

- Instance Properties

  `board.fLength` -- determines the distance, in pixels, from the camera to the center of the bounding cube. The default value: `board.fLength=2000`;

  `board.nMesh` -- gives the grid on the xy-plane which determines the number of nodes. The default value: `board.nMesh=15`. The user can change the value.

`board.nFrameColor` -- determines the color of the wireframe. The default value: `board.nFrameColor=0x333333`.

`board.nAxesFront` -- determines the color of the boxed axes appearing in front of the surface. The default value: `board.nAxesFront=0xCCCCCC`.

`board.nAxesBack` -- determines the color of the boxed axes appearing behind the surface. The default value: `board.nAxesBack=0x666666`.

`board.bShowFrame` -- determines if the wireframe is displayed or not. Boolean. The default value: `board.bShowFrame=true`.

`board.nOpacity` -- determines opacity of the surface drawn in percent. The default value: `board.nOpacity=100`.

`board.aTilesColors` -- an array of colors for each rectangle of the surface. Depends on the grid and the user's choice of color scheme. Intial value: `board.aTilesColors=[];`

- Instance Methods

`board.setBackColor(0xFFFFFF)` -- determines the color of the graphing board background. The color passed as a parameter in hex. If the method is not called, the color of the background will be black by default.

`board.setBorderColor(0x000000)` -- determines the color of the border of the graphing board. The color passed as a parameter in hex. If the method is not called, the color of the border will be white by default.

`board.setErrorBoxFormat(0xFFFFFF,0xFFFFFF,0x000000,12)` -- determines the colors of the Error box background, border, the color of the font, and the size of the font. If the method is not called, the default values are: black, black, light gray, 12.

`board.setErrorBoxSizeAndPos(290,100,20,30)` -- determines the width, the height, and the x and y coordinates (all in pixels) of the Error box. The position is with respect to the upper left corner of the graphing board. If the method is not called, the default values will position the Error box over the top half of the graphing board with black background and border, and light gray text of size 12.

`board.setErrorBoxVisible(true)` -- sets the visibility of the Error box.

`board.setWaitBoxFormat(0x009900,12,"Processing…")` -- determines the colors of the Wait box font, the size of the font, and the text of the message. If the method is not called, the default values are: orange, 12, "Processing...".

`board.setWaitBoxSizeAndPos(80,20,240,5)` -- determines the width, the height, and the x and y coordinates (all in pixels) of the Wait box. The position is with respect to the upper left corner of the graphing board. If the method is not called, the default values will position the Wait box in the upper right corner of the graphing board.

`board.setWaitBoxVisible(true)` -- sets the visibility of the Wait box.

`board.setLabelsFormat(0x000000,10)` -- determines the colors of the font and the size of the font for xmin, xmax, x etc. labels appearing at the corners of the bounding cube. If the method is not called, the default values are 0xCCCCCC and 11.

`board.setCubeSize(50)` -- determines the size of the bounding cube formed by the boxed axes. If the method is not called, the size will be set at approximately one third of the graphing board size.

`board.getCubeSize()` -- returns the size of the bounding cube.

`board.getBoardSize()` -- returns the size of the graphing board (which was passed to the constructor).

`board.drawSurface(M)` -- renders the new view of the graph of a function determined by the rotation matrix M (where M is an array passed as a parameter). The method rotates the surface by M from the surface's last position before calling the method, not from the initial position of the graph.

`board.drawAxes(M)` -- renders the new view of the bounding cube determined by the rotation matrix M (where M is an array passed as a parameter). The method rotates the bounding cube by M from the cube's last position before calling the method, not from the initial position of the cube.

`board.showError(mes)` -- makes the Error box visible and displays the text given by the parameter mes. The parameter mes must be a string.

`board.resetBoard()` -- erases all graphs, makes all Wait and Error boxes invisible, resets board.pixArray=[] and board.funArray=[].

`board.destroy()` -- removes all movie clips, text fields created by the instance. The method should be called before deleting a variable in which an instance of GraphingBoard3D is stored (should you ever need to do so).

- **MatrixUtils**

This class contains four static methods which perform basic matrix operations that we need in our script. These methods are:

`MatrixUtils.MatrixByVector(A,B)` -- multiplies a matrix A by a vector B.

`MatrixUtils.MatrixByMatrix(A,B)` -- multiplies a matrix A by a matrix B.

`MatrixUtils.projectPoint(point,flen)` -- projects a point onto the 2d screen with the camera distance flen. (In our case, flen=board.fLength.)

`MatrixUtils.rotMatrix(a,b,c,theta)` -- calculates the rotation matrix about the axis (a,b,c) by an angle theta in degrees.

- **ColorUtils**

This class contains the static method which, given a grid and the user's choice of a color scheme, returns an array of colors passed to "board" as board.aTilesColors:

`ColorUtils.setUpSquareColors(grid,choice)` -- grid is the current board.nMesh, choice is an integer from 1 to 6.

The name of the method "setUpSquareColors" reflects the way the array of colors is created. It assigns a color to each rectangle on the grid of rectangles on the xy-plane and does not take into the account the value of a given function at any point of each rectangle. Such a method is suitable for coloring graphs of functions of two variables. It is not suitable for coloring parametric surfaces. In the case of parametric surfaces, coloring has to be based on the position of each rectangle in the 3d space. In the upcoming article by the authors about parametric surfaces, the class will be augmented by setUpPositionColors(…,…) method.

Changing Functionality

Any significant changes in functionality of the grapher require delving deeper into the code. The most desirable addition to the functionality would be adding the "scaling constrained option". This is not difficult and can be accomplished by changes and additions to the prepGraph( ) function and the xtoPix, ytoPix, znd ztoPix functions. It is a good topic for an initial tutorial.

Recommended Reading

The complete Flash 8 documentation, including Getting Started with Flash, Using Flash, Learning ActionScript 2.0 in Flash, ActionScript 2.0 Language Reference, Using Components in Flash, and more can be downloaded for free from the Adobe/Macromedia site:

http://www.adobe.com/support/documentation/en/flash/

(After you open the page, click on the link Download Complete Flash 8 Documentation.)

A few books that we have found immensely helpful are listed below.

To become familiar with the Flash's authoring environment:

1. Katherine Ulrich, Macromedia Flash 8 for Windows and Macintosh: Visual QuickStart Guide, Pearson Education, 2006.

2. Robert Reinhardt and Snow Dowd, Macromedia Flash 8 Bible, Wiley Publishing, 2006.

To learn ActionScript 2.0 (which hasn't changed between Flash MX 2004 and Flash 8 except for some classes and functions added):

3. Colin Moock, Essential ActionScript 2.0, O'Reilly, 2004.

4. Macromedia Flash MX 2004 ActionScript 2.0 Dictionary, Macromedia Press, 2003.

5. Robert Reinhardt and Joey Lott, Flash MX 2004 ActionScript Bible, Wiley Publishing 2004.

6. Colin Moock, ActionScript for Flash MX: The Definitive Guide, Second Edition, O'Reilly, 2003.

The latter book was written for an earlier version, Flash MX, but we still find it to be a great reference.