

An Introduction to Creation of Interactive Virtual Models with MuPAD and JavaView

Miroslaw Majewski

Zayed University, United Arab Emirates

e-mail: majewski at mupad.com

Abstract: In this paper we will show how one can produce mathematical graphic using a Computer Algebra System such as Maple, MuPAD or Mathematica and transform it to an interactive virtual model for online display and interactive experimentation with JavaView.

Introduction

Modern Computer Algebra Systems, in short CAS, are able to produce high quality graphics and animation. However, while converting such graphics to web acceptable formats, users end up with large files, bad quality, and mostly static images. Graphics formats like GIF, JPG or PNG are not an option if we wish to produce interactive online materials and allow students some interactive experimentation. A good choice for developing mathematical plots for online display is JavaView. In fact, we will obtain much more than plots. With JavaView we will be able to produce interactive mathematical models, in this paper often called Virtual Models or in short VM, that allow users a very sophisticated experimentation.

From a user point of view this is important to point out how such interactive experimentation may look. Imagine that we open a web page with a Virtual Model of a complex polyhedron or a surface obtained from parametric equations. An interactive experimentation with such model may include such simple operations like rotations of the model or zooming in into it to analyze particular fragments of the model. A more sophisticated experimentation may include such operations like isolating and removing particular parts of the model, simplifying the model, applying specific transformation matrices, projecting model onto a sphere or another surface, creating normal vectors for the model surface, and many other operations known from geometry, differential geometry or topology. Finally, we can think about modeling operations and applying special effects allowing us to produce stellated surfaces, or unfolding the surface into a flat mesh of polygons.

Virtual Models can play a serious role in teaching as well as in scientific experimentation. However, in this paper we will concentrate only on creating such models and enhancing their visual representation. In this paper, we will discuss all major aspects of creating such models: the creation of a graphic using a CAS, exporting it into JavaView acceptable formats, enhancing the functionality of a model in JavaView and finally optimizing such a model for online display.

For the purpose of this paper, we will base our investigations on graphics created in MuPAD. At the time of writing this paper, MuPAD graphics are the most sophisticated graphics one can produce using CAS. Maple or Mathematica users can find some relevant information on Maple and JavaView web sites (see [1, 7] for Maple, and see [5, 6] for Mathematica). Detailed information about MuPAD graphics can be found in books [3, 4]. Finally, examples of Virtual Models using JavaView can be found on JavaView web site (see [8]). The Plus Magazine and MathPAD Online published a few papers using JavaView to display mathematical surfaces (see [9, 13]). A collection of Virtual Models using JavaView can also be found on EG-Models and GANG web sites (see [10, 11]). Finally an extensive showcase of MuPAD graphics using JavaView is published on Virtual Models research project web site (see [12]). Recently, Zuse Institute and a few Balkan universities conducted large European Union project on Multimedia Technology in Mathematics and Computer Science. As a part of this project an online analytic geometry course was developed at Belgrade University. The course uses JavaView to display examples of 2D and 3D geometry (see [14]).

How JavaView Works

JavaView is a computer application written using the Java programming language. Users can use JavaView on a local computer in exactly the same manner as any other executable program, or through a web browser as an applet web page plug-in. At this stage it is essential to mention that, in order to use JavaView, neither the developers nor the end-users need to know how to program in Java. However, for developers of VM a basic knowledge of programming in Java can be very useful.

In order to display VM, we need to produce a model in one of the formats that JavaView understands, e.g. JVX, JVD, MPL, MGS, or some other popular file formats. Later we can open such model through the JavaView menu or link the model and JavaView applets to a web page. In the second case we will be able to publish such model on a web site and display it online. Note, JavaView can also be directly called from inside some CAS which avoids of creation of intermediate files.

How CAS Compute Graphics

The way that CAS produce mathematical graphics differs very much from the way graphics are rendered by specialized graphical programs like 3D Studio, Maya or Strata 3D. In the first case, we need to obtain a picture in reasonably short time. Therefore, some simplified methods and special algorithms are used. We obtain a graphical object constructed out of points, line segments, and flat patches. Due to the fairly simple structure of such models, we are still able to manipulate the model inside the CAS. In most examples, we deal with plots created from the formula of a function. In some rare cases we deal with plots obtained from implicit equations.

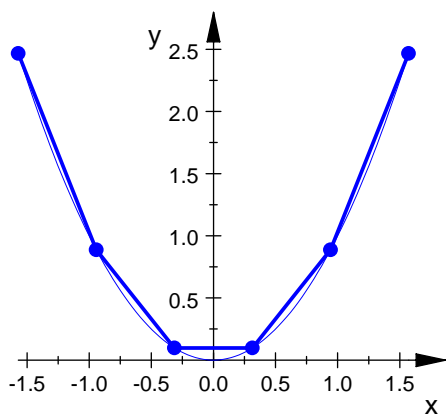


Fig. 1 Mesh representing a graph

While producing a plot of a function, any CAS divides the domain of the function into a number of equal segments for functions of one variable, and into rectangles for functions of two variables. This way we obtain a one- or two-dimensional grid. In the next step, the CAS calculates values of the function for each point of the grid. Using the obtained values, the CAS produces a mesh representing the function. Finally, the mesh is displayed on the computer screen.

It is important to notice that such an image looks like the graph of our function but, in fact, it coincides with the real graph of the function only in a finite number of points (see fig. 1). In the presented figure, the thin line represents the real graph of the function while the thick line is the graph that we usually obtain in CAS. The marked points were obtained from calculated values of the function.

An important conclusion of the above-mentioned fact is that denser grids produce more accurate representations of functions. The natural consequence of this fact is that very accurate plots result in larger files than those obtained from loose grids. Therefore, in the future we will have to choose whether we wish to get a more accurate graph and a larger file or a less accurate graph but a smaller file (see fig.2a, 2b, 2c).

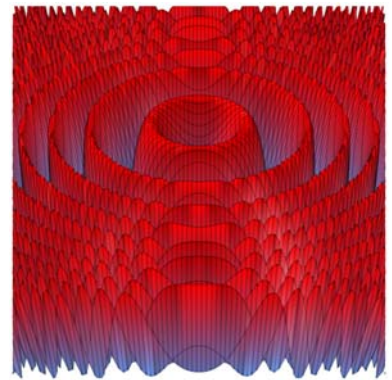
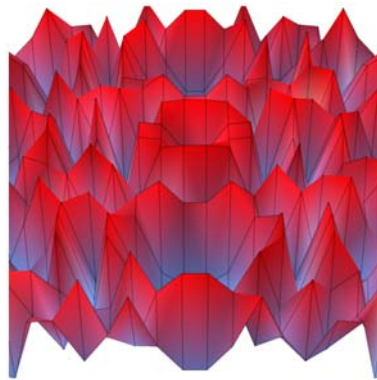
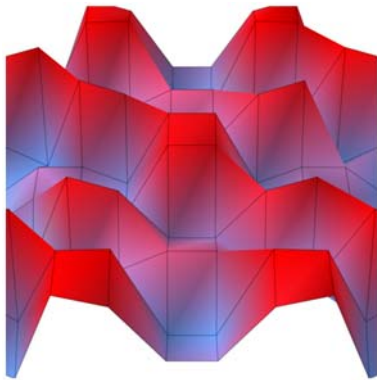


Fig. 2a – grid 10x10, JVX file 12KB

Fig. 2b – grid 20x20, JVX file 45KB

Fig. 2c – grid 100x100, JVX file 1,2MB

File Formats

Each of the earlier mentioned CAS – MuPAD, Maple and Mathematica, has a way to save its plots in formats acceptable by JavaView. In this paper, we will concentrate on two JavaView native formats, JVX and JVD. Both formats are XML based and can be edited using any text editor as well as using XML-dedicated editors like XMLWriter or Cooktop. The most important feature of XML files is that with a minimum of effort, the contents of these files can be read, understood and modified by an average VM developer.

The JVX file contains description of a model while the JVD file contains description of the environment – camera, lights, background and some other parameters. A typical entry, so called element, in JVX or JVD file contains two tags (starting tag and closing tag), the information enclosed between these tags and declarations of parameters relevant to the given element. Entries in XML files can be nested, for example a declaration of lights may contain a number of lights, each with a specified type and properties. In JVX/JVD files multiple numerical parameters are separated by the space character, and their format is based on well-known notions from computer graphics. For example, the color of a light can be declared using the RGB concept with the amount of each component ranging from 0 to 255. Below we show a fragment of a JVD file describing a set of lights for the plots shown in figure 2.

```
<lights lightingModel="Light">
  <light type="ambient">
    <color>255 255 255</color>
    <intensity>0.125</intensity>
  </light>
  <light type="point">
    <position>15.708 18.8496 -7.99984</position>
    <interest>0 0 1.71064e-005</interest>
    <color>255 255 255</color>
    <intensity>0.4</intensity>
    <exponent>50</exponent>
  </light>
</lights>
```

As we said before, the JVX file contains description of a model. The essential part of the file contains description of each geometry used in the model. For example, a surface given by a formula is considered as geometry. Each geometry declaration may contain a list of points forming nodes of the mesh, information about which nodes are connected to obtain the faces of the mesh, colors of points or faces, information which points are neighbors, etc. Therefore, the more accurate the model, the larger its JVX file is. Below we show a fragment of the JVX file describing the surface shown in figure 2a.

```
<pointSet dim="3" color="show" point="hide">
  <points num="100">
```

```

    <p>-3.14159 -3.14159 -0.405037</p>
    <p>-2.44346 -3.14159 0.226945</p>
    ...
    <p>3.14159 3.14159 -0.405037</p>
    <thickness>2.65748</thickness>
</points>
<colors num="100">
    <c>146 105 167</c>
    <c>195 57 91</c>
    ...
    <c>146 105 167</c>
</colors>
</pointSet>
<faceSet face="show" edge="show" color="show" colorSmooth="show">
    <faces num="81">
        <f>0 1 11 10</f>
        <f>1 2 12 11</f>
        ...
        <f>88 89 99 98</f>
        <color>255 0 0</color>
    </faces>

```

Observe that each element of the JVX file contributes to the size of the file. Therefore, in many situations we may sacrifice some features of the VM in order to obtain a smaller file. For example, a good practice is to use a less sophisticated coloring scheme, less accurate numbers for describing coordinates of points, etc. For instance, in many cases it would be enough to specify the point coordinates using two decimal digits instead of ten decimal digits. On a computer screen, users may not notice any significant difference between the points `<p>3.141593456 3.141597645 -0.4050379864</p>` and `<p>3.14 3.14 -0.40</p>`, but the JVX file will be significantly smaller. Some of the mentioned optimizing operations should be taken into consideration while obtaining the plot in the given CAS, while some others can be done later when editing the model in JavaView.

Selected Technical Considerations

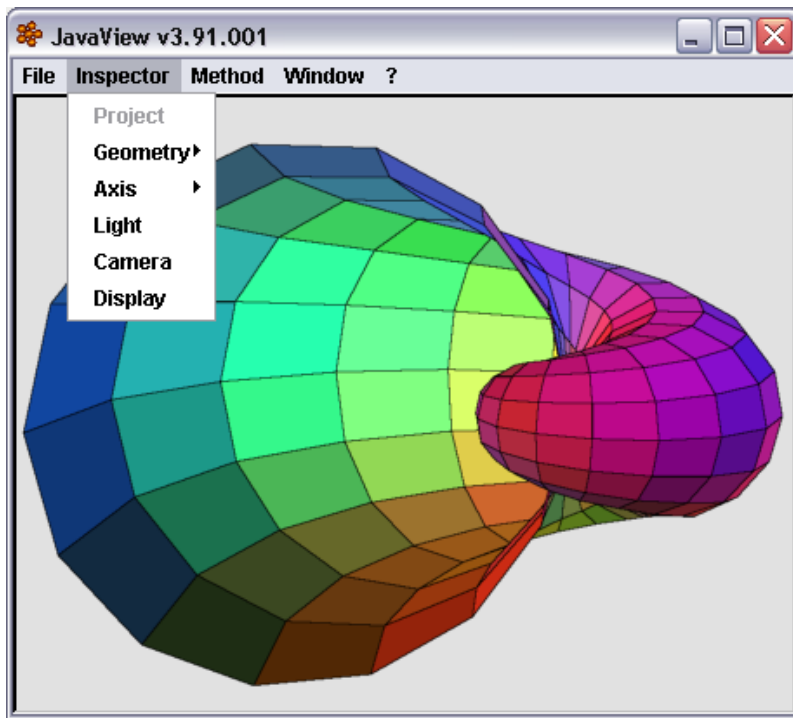


Fig. 3 JavaView screen

few JavaView functions related to coordinate axes, lights, camera and display (see fig. 3).

JavaView can be downloaded from JavaView web site (see [8]), and run on a local computer by executing the `javaview.exe` file (on Windows only) or by executing the `javaview.bat` file.

Coordinate Systems

For a number of models, we may not need a coordinate system at all. For instance, when developing virtual models of polyhedra we may not need a coordinate system at all. However, for models of surfaces or curves in 3D, the coordinate system may play an important role in locating the model in the space and allowing us to read the coordinates of some elements of the plot. In JavaView the concepts of coordinate systems may differ slightly from those in CAS. However, we still can choose between the traditional, so-called origin-based coordinate system, or framed and boxed coordinates. We can assign colors to the coordinate axes and declare their thickness. Finally, we can add to the coordinate planes a very precise grid.

Another important issue while developing VM is scaling. Most, but not all, CAS use automatic, sometimes called unconstrained, scaling of each axis of the coordinate system separately, to enhance the readability of the graph (see fig. 4a, 4b). However, in CAS we never have a chance to define our own scaling. In JavaView we can precisely define the scaling for each axis of the coordinate system.

There are a number of issues that are usually not considered while obtaining plots in a CAS. Moreover, some CAS do not have means to address some of these issues. For example, in MuPAD we can produce antialiased plots while in Maple, Mathematica or Derive, we obtain plots in pure vector formats. Most CAS do not have implemented a light concept, functional coloring or transparency. Fortunately, most of these features can be added or changed while editing the VM in JavaView. It is important to mention that JavaView offers much more than only enhancing our models. There are a number of algorithms implemented in JavaView to perform some mesh operations or functions necessary for differential geometry, and many others. Most of these operations are available under the **Method** menu. For the purpose of this paper, we will concentrate on a

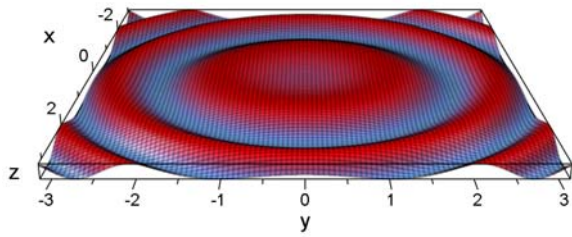


Fig. 4a Plot using constrained scaling

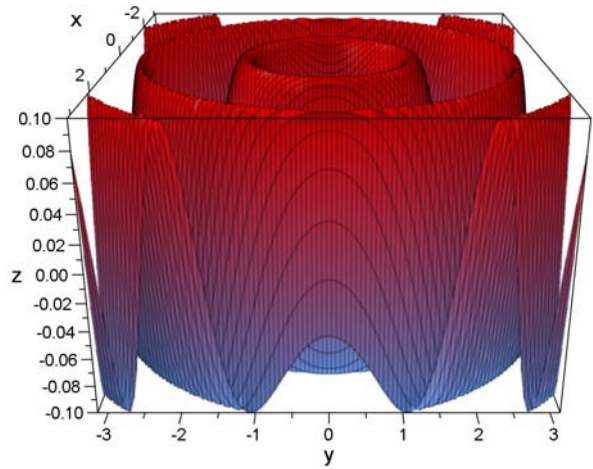


Fig. 4b The same plot using automatic scaling

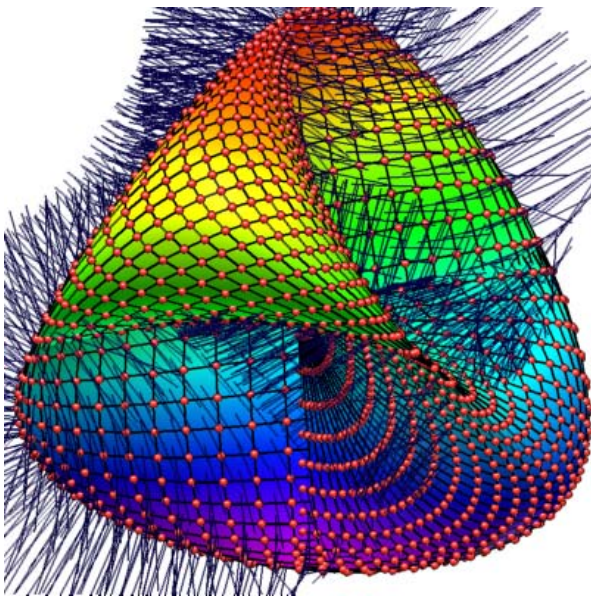


Fig. 5 Hairy Steiner surface obtained after material modifications in JavaView

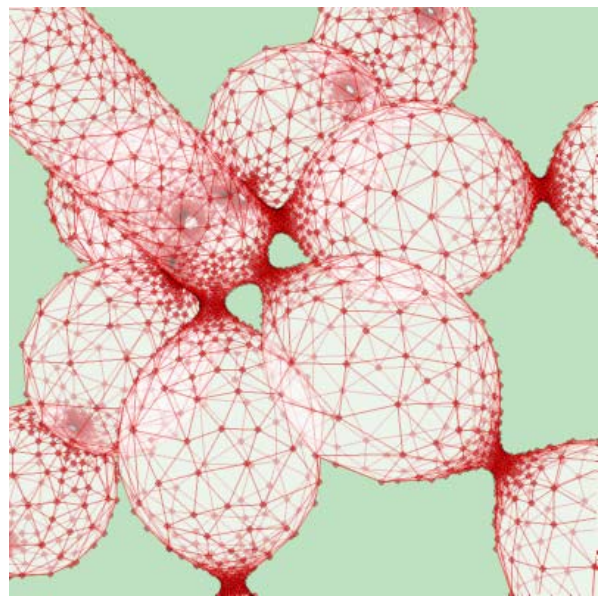


Fig. 6 Transparency helps us to show the internal structure of a complex object

Material Properties

Material properties (JavaView menu **Inspector>Geometry>Material**), is usually the place where most of JavaView developers and users start their experiments. This is where we can show or hide the vertices of the model, the edges, faces or normal vectors (fig. 5). We can assign colors to them or add transparency (fig. 6). This is also the place where some artistic features can be added to our model. Figure 5 shows a hairy Steiner surface with mesh nodes represented as small balls, and face normals displayed. Figure 6 presents a complex mathematical object where transparency was used to show the internal structure of the object.

Using Lights and Cameras

The majority of mathematical graphs use flat ambient light. However, using a specific light type may significantly enhance the scene and our models will look more interesting. Most CAS do not have the light concept implemented at all. At the time of writing this paper, MuPAD is the only CAS where various types of lights were implemented. There is quite good match between the light types in MuPAD and JavaView. Table 1 shows the mapping between MuPAD and JavaView lights.

MuPAD	JavaView	Description
Ambient light	Ambient light	Both light types are identical.
Point light	Point light	Point lights in both systems are almost identical.
Spot light	Spot light	The spot light in JavaView is a new type of light implemented recently (ver. 3.92, June 2005). It contains a number of features that are not available in MuPAD.
Distance light	Direction light	The distance light and the direction lights share the same features; when exporting the JVD file the distance light is saved as a direction light.
	Head light	This is a unique JavaView light. This light is attached to the camera and does not move when rotating the model. It behaves like a direction light.
	Sky light	This is another unique JavaView light. It simulates sunlight and enlightens the scene from the top. Its properties are similar to the direction light.
<i>Table 1 – Comparison of light types in MuPAD and JavaView</i>		

MuPAD, by default, in each scene uses six point lights located around the scene, and one ambient light. Three of these lights are stronger, and the remaining three are used to complement the main lights. In JavaView the default set of lights contains two lights only – the head and the sky lights. Users can use another predefined set of lights, the so-called RGB lights with three lights: red, green and blue. Finally, in both systems users can declare their own lights. Adding lights to a virtual model may not be a very straightforward task. For many models, we have to experiment with different types of lights and their location in order to better display specific parts of the model. Figure 7 shows the same model with two different sets of lights. The model on the left picture uses the default for many CAS flat lighting. The model on the right picture uses three spot lights.

At this point, it is important to mention that the more lights are used in our VM, the more calculations JavaView has to perform. Consequently, manipulation of the VM in JavaView may slow down. Therefore, for many models the six default lights obtained from MuPAD should be replaced by fewer lights.

While developing a VM, it is important to establish the correct view of the scene, perspective, transformation of the space and projection. Both MuPAD and JavaView allow us to declare a sophisticated camera, and view our VM through multiple cameras. In MuPAD we create a camera by declaring its location, the point in the space where the camera is pointing, the angle of the lens, and a few other parameters. While transforming a MuPAD plot into JavaView model, the camera declaration is placed in the JVD file. All camera parameters are exported in such a way that the model in JavaView looks exactly the same as it did in MuPAD. However, often it is more convenient to modify camera properties in JavaView. In JavaView we can adjust most of camera properties by visual manipulation. Moreover, we can project our model into spherical or hyperbolic space.

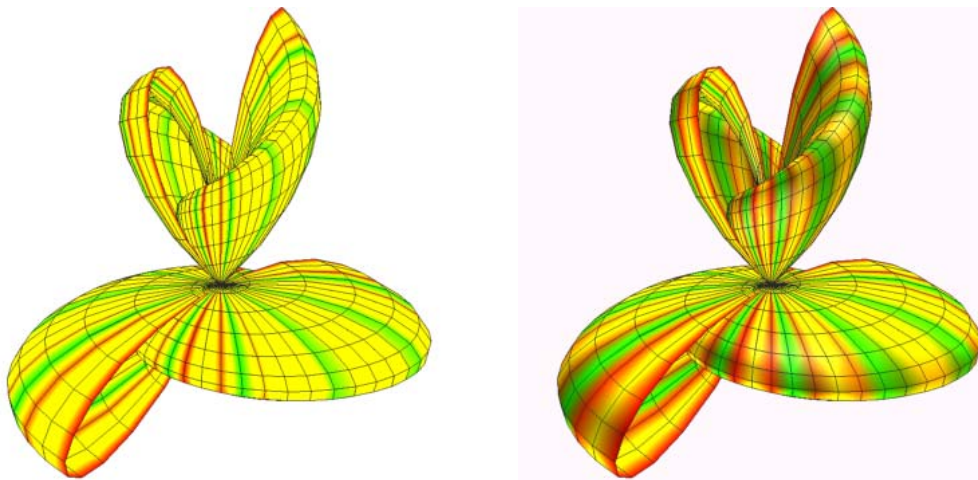


Fig. 7 Lights may significantly enhance the model (left – flat light, right – three spot lights)

Other Important Parameters

While developing realistic models in 3D space, we have to solve a number of issues related to how 3D graphics are displayed on the computer screen. Some of them are: antialiasing, sorting model faces before displaying it, simulating the depth of the scene, or using the z-buffer for processing complex models. Let us concentrate a while on antialiasing. Most CAS display graphics on the computer screen in a ragged form where slanted lines are shown as objects built out of small rectangles. In order to improve such a display, we use special technique known as antialiasing that makes such lines smooth. At the time of writing this paper, MuPAD is the only CAS that uses antialiasing while displaying 3D plots. In JavaView we also have an opportunity to switch antialiasing on and off. Figures 8a and 8b show how antialiasing works in JavaView.

Although antialiasing improves significantly the display of 3D graphics on the computer screen we should use it carefully. Antialiasing of a complex model is a very time consuming task. Therefore, using antialiasing in some Virtual Models we may slow down manipulations in JavaView, although JavaView disables antialiased rendering during user interaction.

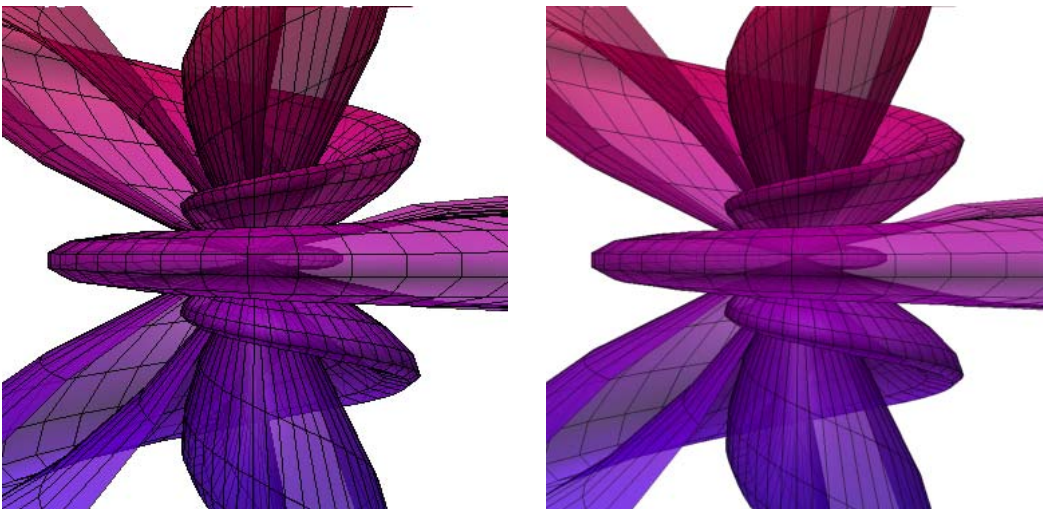


Fig. 8a VM in JavaView, antialiasing is off *Fig. 8b VM in JavaView, antialiasing is on*

Improving Performance

While developing Virtual Models, we have to take under consideration some limitations that the users of our models may face. These are the bandwidth of the user's connection to the Internet, and the power of the user's computer. Therefore we should optimize our models in many ways.

One of them, not necessary the most important, is reducing the size of JavaView model files. JVD files are usually very small. Therefore, we have to concentrate on JVX files. As we said before, using numbers with a smaller number of digits may influence the size of the file. Another element is the white space added while formatting JVX file in XML editors. For example, adding one space or tab character in each line of a file with 10,000 lines will increase the size of the file by 10,000 bytes (about 10KB). JavaView contains a number of tools that may help us in optimizing the model and making its file smaller. We can remove unnecessary or duplicate elements, remove unnecessary colors, simplify colors, or combine a number of objects in one. Each of these operations may result in much smaller files. For example, by removing individual colors for each vertex and applying a global color scheme, we may end up reducing the size of the JVX file even by about 30%. Finally, the JVX file can be compressed and saved as **myfile.jvx.zip** which effectively compresses most of the redundant blanks and tabs.

Optimizing the size of the model file will definitely help us while downloading models through the Internet. However, sometimes a more complex model can freeze JavaView while performing experiments with the model. By turning on a few special features to improve the display of the model on the computer screen, we may overload JavaView's computing engine or even freeze it for some time. However, we can always choose between rendering speed and quality of the image. The most time-consuming features to calculate are: antialiasing, enabling the 3D look for flat objects, simulating depth of a scene, or projecting a model into a non-Euclidean space.

Future Considerations

In this paper, we presented the most fundamental aspects of producing Virtual Models. JavaView offers a number of tools allowing us to transform models obtained in CAS. It is especially worthwhile to explore the set of tools to produce special effects like stellate, punch, or unfold geometry. Figure 9 shows Kuen surface, with a backdrop of the Earth seen from the NASA space shuttle. The final VM was modified using the punch effect from **Methods>Effects** menu in JavaView.

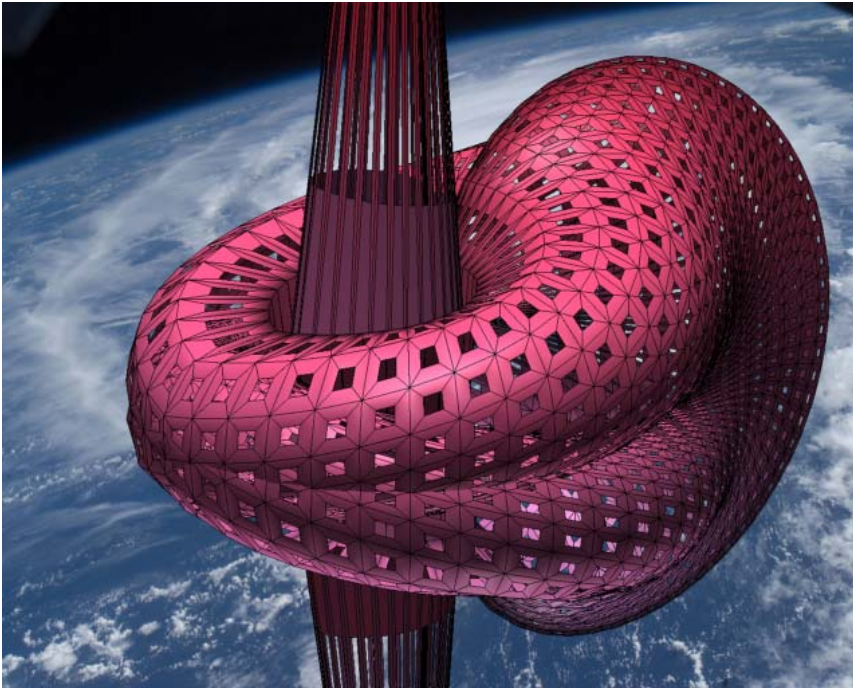


Fig. 9 Realistic model of Kuen Surface produced in MuPAD and JavaView

In the figure 10 we show how transparency and magnified vertices allow us to see the structure of the surface. The twizzler is a very complex surface with constant mean curvature. Its internal structure is even more complex than this what we see from outside. In order to see its interior we used camera clipping. This way we were able to bore two large holes in the surface opposite the camera. Moreover, rotations of the model in JavaView, or fly through the surface in JavaView, may give us a detailed look into its interior.

JavaView in many ways proved its value in investigating various problems in 2D and 3D geometry. Visualization of mathematical objects in JavaView brings new experiences and new knowledge in teaching such disciplines like analytic geometry, topology, differential geometry or even mathematical art. Virtual Models may also play an important role in primary or high school teaching. For instance we may think about developing an online showcase of various types of polyhedra, polynomial surfaces, L-systems, etc.

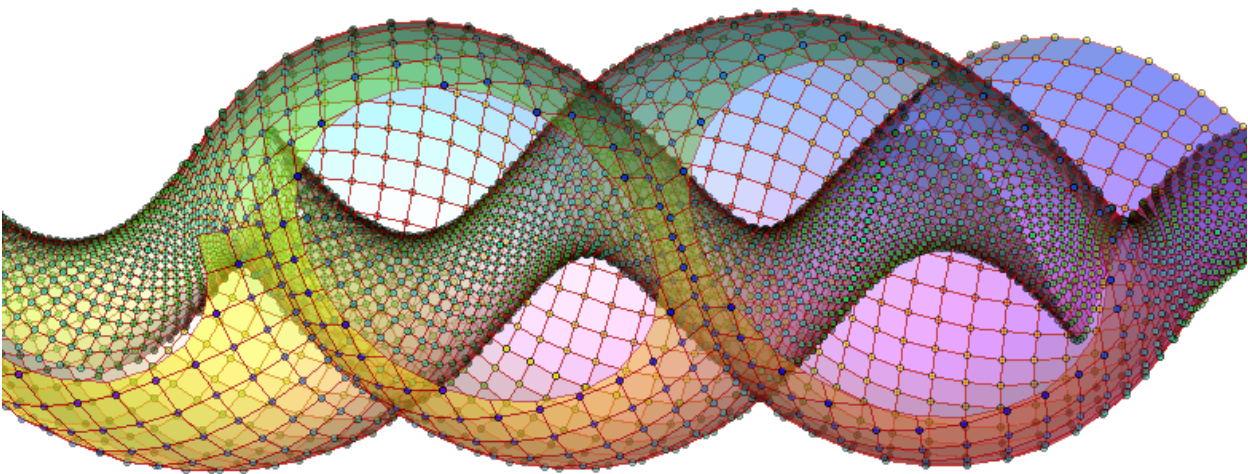


Fig. 10 The twizzler, a constant mean curvature surface in JavaView

References

1. S. Dugaro., K. Polthier: *Visualizing Maple Plots with JavaViewLib*, published in Algebra, Geometry, and Software Systems, ed. M. Joswig, N. Takayama, Springer Verlag, 2003, pp.255-275.
2. M. Majewski: *MuPAD Pro Computing Essentials*, Springer Verlag, Berlin-Heidelberg, 2004, pp. 538.
3. M. Majewski: *Getting Started with MuPAD*, Springer Verlag, Berlin-Heidelberg, 2005, pp. 270.
4. M. Majewski, K. Polthier: *Using MuPAD and JavaView to Visualize Mathematics on the Internet*, Proc. of the 9th Asian Technology Conference in Mathematics, 2004, pp. 465-474.
5. K. Polthier: *Mathematica and JavaView*, 2002, <http://javaview.zib.de/mathematica/>
6. K. Hildebrandt, K. Polthier: *JavaView & webMathematica*, 2005, <http://mathematica.zib.de/>
7. K. Polthier: *JavaViewLib - A Maple Powertool*, <http://www.javaview.de/maple/>

Web Sites using JavaView

8. JavaView web site, <http://www.javaview.de>
9. +Plus online magazine: <http://plus.maths.org>
10. EG Models: <http://www.eg-models.de>
11. Geometry Analysis Numerics Graphics (GANG): <http://www.gang.umass.edu>
12. Virtual Models research project web site: <http://majewski.mupad.com/mjv/>
13. MathPAD Online: <http://www.mathpad.org>
14. Analytic Geometry course at Belgrade University: <http://codd.matf.bg.ac.yu/angeom/>