A Scientific Visualization Synthesizer

Roger A. Crawfis (crawfis@llnl.gov)
Michael J. Allison (allison4@llnl.gov)

Lawrence Livermore National Laboratory
Livermore, CA 94551

Abstract

We describe methods for displaying scientific data using textures and raster operations rather than geometric techniques. The flexibility and simplicity of raster operations allow a greater choice of visualization techniques with only a small set of basic operations. In addition, texture mapping techniques will be shown that allow the representation of several variables simultaneously, without a high degree of clutter. The combination of traditional geometric techniques, image composition techniques and image rendering techniques can be integrated into a single framework for the display of scientific data. This paper presents a system for generating and operating on textures and images for the purposes of scientific visualization. The advantages of using such a system are demonstrated through the use of examples. In particular, the development of bump maps for vector filters and contour lines is demonstrated.

Introduction

There has been significant press over the past several years on the benefits of working with several individual images and combining these to produce realistic images [Porter 84], [Cook 84], [Hanrahan 90a], [Haeberli 90]. These same arguments can be applied to the field of scientific visualization, where instead of compositing several images of rendered objects together, we wish to composite several graphs, images or plots. The principles are the same; we wish to overlay a mesh or several contour curves on top of an existing graph. This overlaying can be accomplished by plotting the various components in a specified order. However, by blending the raster representations of these graphs together, images can be improved to highlight the most pertinent information and provide more subtle cues for less important information.

Recent research in volume rendering has demonstrated the potential for direct rendering of information. This allows large data sets to be rendered without mapping the data to a large set of geometric objects. With the size of data sets rapidly growing, problem definitions are quickly being defined on grids of resolution approaching that of typical graphics workstations. Drawing lines, polygons or vectors at every mesh point may lead to a very cluttered image. Continuous coloring or data mapping, on the other hand, provide a smooth representation of the data. The use of textures can be extended to include vector fields and contours as well.

This has motivated the development of an experimental system and language for the interactive creation of textures and raster images. The language facilitates the manipulation of images and textures by allowing the user to deal with them as individual objects. An interactive interpreter for this language has also been developed to allow interactive experimentation of various texture mapping schemes.

We will first describe the Texl language and interpreter. Then we will discuss how one can map discrete scientific data to various textures or maps that can be combined in the rendering process. This will be done through the extensive use of examples. We will restrict the majority of our attention to two dimensional surfaces with full three-dimensional renderings, and briefly show how these techniques can be extended into three dimensions.

The Texl Texture Synthesis System

In dealing with texture maps and the data with which to derive texture maps we determined that a flexible manipulation system was required. Some of the goals we desired include programmability and extensibility. While a standard programing language provides all of the flexibility one might need, the edit, compile, and link cycle was too long.

Other systems, used for similar work, provide the flexibility within a hybrid interpreted and compiled environment. The "Image Synthesizer" [Perlin 85] creates an environment that allows complete programmability over the color of individual pixels. The synthesizer also introduced the concept of solid or 3D textures. The concept of a generalized shading language to be used by rendering systems was introduced by [Upstill 89] and [Hanrahan 90b]. By extracting features from the previous systems, we were able to construct a system to be used for synthesizing textures.

Our system, the Texl language, was designed to be similar to C for control structures. Data types are fixed and geared towards pixels, vectors, and texture maps. Since some applications may require arrays, the texture maps are allowed to be variable in size and depth to allow them to be used as numeric arrays. All numeric operations are the same as the C language. A summary of data types and operations used in the system follows.

Texl Data Types and Operations

Color	+, -, *, /, =, composition operators
Point, Vector	+, -, *, /, =
Number	Normal numeric ops, C math library
Surface, Volume	+, -, *, /, composition operators, C math library

Higher level constructs, such as colors or vectors, are allowed in expressions. A user may add colors via the plus ("+") operator, or combine them using other operators. In some cases scalar and vector quantities are allowed to be mixed. A user may scale a color, or vector, using the following expression:

scalar * [1, 0, 1]

For maximum flexibility, types of similar representations may be mixed. For instance, an expression may add a point to a vector yielding a vector. All data types in the system are considered to be first class objects. This allows the programmer to create variables of any named type and assign values of expressions to the variables. As in C, assignment is an operator which produces a side effect. The assignment operator may be used within expressions, such as control expressions in `"for"` or `"while"` loops.

All colors are specified to contain 4 components: red, green, blue, and alpha. The alpha component specifies the opacity of the indicated pixel. Alpha channels were introduced by [Porter 84] for the purposes of compositing digital images. Compositing operators are provided for individual pixels or complete images. In addition to allowing textures to contain colors, they may also contain a set of vectors, v ^{n}. We have used them to contain multiple scalar values of data, which are later transformed via some function to a texture. Scientific or engineering data are imported to the system and appear in a texture variable.

For better efficiency and quicker running speed, an external routine option is implemented. Often used or complex routines can be compiled into the system and are callable as if they are Texl routines. This allows such routines to perform at high rates of efficiency. External routines have been implemented to allow use of the standard Unix™ math library. We have also implemented other routines that can benefit from being compiled in C such as three dimensional noise and texturing routines, operations on entire surfaces, and filtering. Due to the similarity of Texl and C, conversion to an external module is usually very easy.

An example of a useful external routine is the `"stamp"` operation. By creating a small (10-30 pixels square) texture, it can be used to indicate aggregate behavior in a data raster. By stamping the small texture, a field of these icons are produced which will provide information on the whole data set. Various parameters of the icon may be controlled such as: size, rotation, or the amount of transparency. Since the icon is another texture, it may be created with Texl, or imported as with any other image or data set.

The system allows the user to load various Texl code files, and select and execute desired functions. Return variables from functions (usually texture rasters) can be stored in global variables. The user may then write the textures to disk or display them on the workstation screen. Rasters may be read in to be used as parameters to a Texl function.

Multivariate Representations

Color shaded contour plots have become commonplace for scientific environments. With color workstations now readily available and affordable, the use of color for scientific visualization needs to be expanded. Simple raster operations provide a fast and flexible way of combining traditional techniques into a single image.

Simple Plot

We will start by showing a sample of typical geometric based techniques: a color shaded or data mapped image, a mesh plot or grid, several isocontour lines and a vector plot. Figure 1 illustrates an image generated using simple raster operations to accomplish these tasks. Note that the grid is obstructed by the vector plot in this image and is somewhat difficult to see. This image was

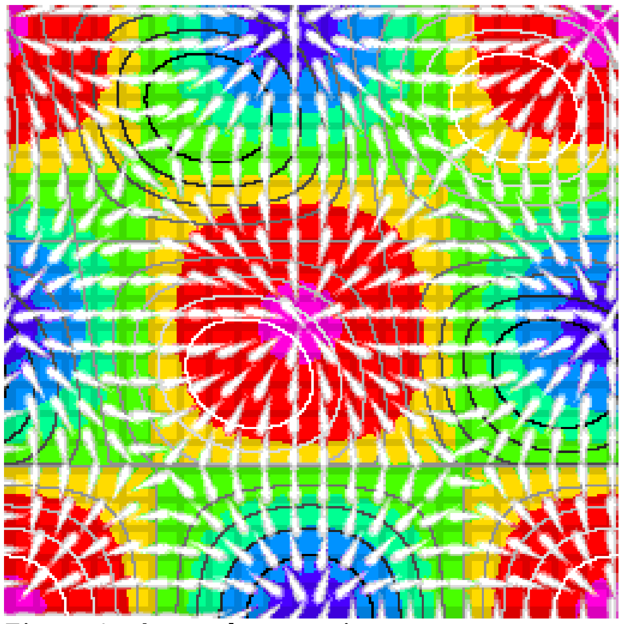
Figure 1. A sample raster image.

```
//********************************************************
********
//   Vector_Plot -- Stamp a vector into a surface at the specified points
//                                   and oriented according to the vector field.
//********************************************************
********
number vector_plot( surface vectors, surface points,
                                   surface stamp, surface out_surf )
{
    number     xs = map_x_size( points );
    number     ys = map_y_size( points );
    number     x, y;
    number     angle;
    number     pi = 3.1415;
    number     vx, vy;
    number     width = 0.1;      //  The vectors are of a fixed size
    number     height = 0.1;     //  occupying 1/10 of the screen image space.
    number     MAX_OP = 2;   //  Stamp the vectors in using the MAX op.
    number     n_vec = 0;

    for( y = 0; y < ys; y++ )
      for( x = 0; x < xs; x++ )
        if( getcmp( points[y,x], 0 ) > 0.0 )   //   Controls where to place
        {                                                       //     the vectors.

//********************************************************
            //  Determine the vector orientation.

//********************************************************
            vx = getcmp( vectors[y,x], 0 );
            vy = getcmp( vectors[y,x], 1 );
            if( vx == 0.0 )
              if ( vy > 0.0 )
                angle = 90.0;
              else
                angle = -90.0;
            else
            {
              angle = 180.0 / pi * atan( vy / vx );
              if( vx < 0.0 )
                angle = angle + 180.0;
            }

            n_vec++;

//******************************************************
            //     Stamp in the vector.

//******************************************************
            stamp_surf( stamp, x/(xs-1), y/(ys-1), width, height,
                                 angle, 1.0, MAX_OP, out_surf );
        }
    print( n_vec; "vectors were plotted\n" );
}
```

Figure 2. Texl code for the vector plotter.

generated by the following simple Texl code:

```
data = normalize( data );
contour = data_map1( data, color_map );
mesh = regular_grid( nx, ny, width, color );
image = max_surf( contour, mesh );
for( i=0; i < n_contr; i++ )
   contour_surf( data2, cont_val[i], cont_color[i], image );

vector = vector_stamp( vector_color );
points = grid_pts( nx, ny );
vector_plot( vec_data, points, vector, image2 );
image = max( image, image2 );
```

Note that the incoming data is sampled into a surface or image. In this particular case, both *data* and *data2* are surfaces with only one channel of information (as opposed to *image* which has four channels for red, green, blue and opacity). The surface *data* is also normalized to lie between zero and one. This allows us to keep our other surface operations simple, but is not required. The data is mapped to a color image using the routine *data_map1*, which simply applies a color lookup on the data. Note that a color map is simply a 1 by N surface with four channels of information. A specialized routine *regular_grid* is called to generate a regular grid consisting of *nx* gridlines in the x domain and *ny* gridlines in the y domain. The width and color of the gridlines can be specified. The mesh generated for this image consisted of grey gridlines on a clear background. This image is then composited with the data mapped image using a maximum compositing operator. By choosing a grey level for the mesh lines, and a standard hot-to-cold color table, consisting of fully saturated colors, the result of applying this compositing operator is to decrease the saturation of the image at the mesh lines.

A contour plot of another data set can be added to the image by using the contour_surf routine. This routine will determine the contour curve for the given value and draw the curve in the specified color. A simple thresholding algorithm was used here.

A vector plot is then added to the image using the *vector_plot* routine. This routine takes as input a vector field (*vec_data*), a description of where to place the vectors (*points*), and a description of the vector to plot (*vector*), and produces as output an image with the vectors composited into it. The Texl code for this routine is shown in Figure 2. The calling syntax for this routine is very simple, yet very powerful. By allowing an image to be passed to the vector plot routine that specifies for each pixel whether a vector should be drawn, an arbitrary placement of vectors can be made. Three point generating routines are provided for this purpose: a regular grid distribution, a uniform random distribution, and a weighted random distribution that takes another data image as input for the weighting. The *vector* argument passed into *vector_plot* is also very flexible, since it specifies an arbitrary image. This allows any image to be used as a vector. In particular, vectors with varying color distributions can be used. The stamp_surf routine shown in Figure 2 is used to map the texture surface onto the vector plot surface. The vector used in Figure 1 was constructed by taking the union of an ellipse with a cone into a 32 by 32 pixel image. The height of this surface controlled the intensity.

To show the generality of this vector plotting routine, we create the image shown in Figure 3 by reading in an image of our LLL logo and using it as the vector in our vector_plot routine. The output of this routine was then

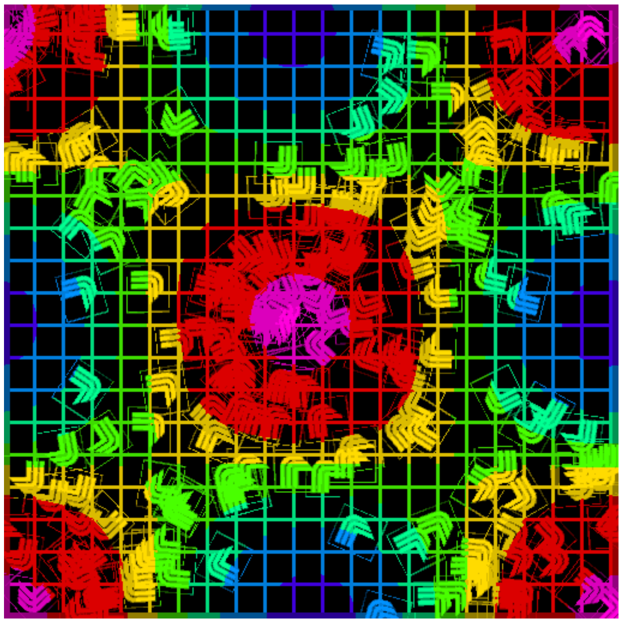
Figure 3. LLL logo used as a vector.

used as a mask over the data mapped image.

Figure 1 portrays large quantity of data, but is too cluttered to adequately convey all of the information. The next example will convey the same information using techniques developed to reduce the overall clutter.

Using Bump Maps to Reduce the Clutter

Since the output of all of our routines are images, we can use those images as height fields of a bump map. In particular, we wish to extract one color component of the images and use that as the height field of a bump map. In Figure 4 we have placed the vector field and the contour curves into a bump map:

```
    data = normalize( data );
    data2 = normalize( data2 );

    contour = data_map1( data, color_map );
    mesh = regular_grid( nx, ny, width, color );
    contour = max_surf( contour, mesh );

    vector = vector_stamp( vector_color );
    points = random_pts( n );
    vector_plot( vec_data, points, vector, bump_map );
    cont_bumps = data_map1( data2, ridge_map );
    bump_map = max_surf( bump_map, cont_bumps );

    spec_map = color_surf( [0.2, 0.2, 0.2, 0.2] );
    image = shade( contour, bump_map, spec_map, light_source
);
```

The vector field used here is essentially the same as that of Figure 1 only a random distribution of the vectors is used rather than a uniform grid of vectors. Since the vector we used in Figure 1 was a smooth surface, it fit well into a bump map representation of the vector field. The contour bumps can be generated using a simple color table mapping where the table is constructed using a sine wave, generating a cycle of ridges. This contour technique has the effect of producing thicker lines or ridges where the data changes less rapidly, hence the gradient of the surface can be seen in the thickness of the contour bumps.

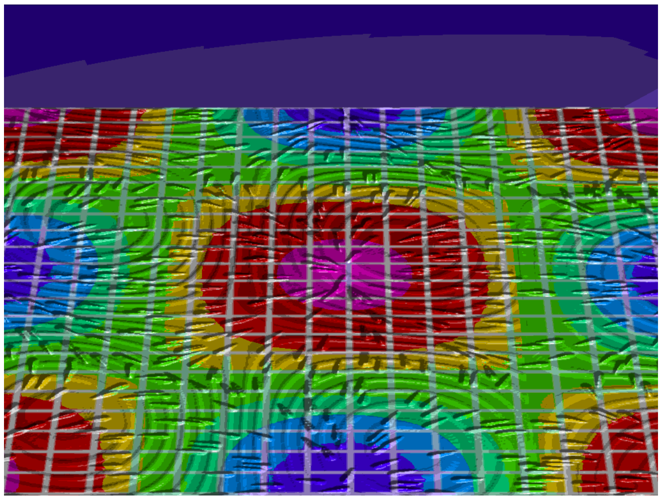

Figure 4. Vector plots and contour plots using a bump map.

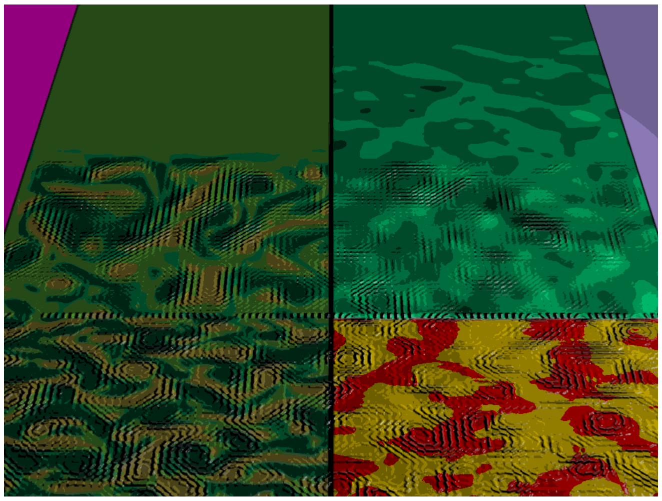

Figure 5. Vorticity, density and velocity fields produced by a shock wave of 2.78 density jump passing through a turbulent field of 0.07%. (Data courtesy of Doug Rotman, LLNL).

Comparing Figure 4 with Figure 1, the cluttering of the image is substantially reduced, and the color data mapping can be easily followed. The choice of which data parameters should be mapped to which representations depends on the degree to which the parameter should be highlighted. Very subtle representations of less important features can beimbedded into the image, where the representation does not disrupt the portrayal of other data, but can be discernable none the less. The mesh plot is more discernable in Figure 4, while the vector field and contour lines are still noticeable, but not quite as apparent. At the same time, the data mapped image is plainly visible.

A Vector Filter

The vector techniques employed above work well for most problems, but can be expensive for large numbers of vectors. For large data sets, or for rapidly changing vector fields, the use of a one pass filter to deposit a vector representation can be both more efficient and more effective. Our first pass at producing a vector filter was to construct a kernel to find the average direction, and then set each pixel in that kernel to the value of a four-dimensional function, f(x,y,vx,vy), where x and y are the pixel locations and vx and vy specify the vector field at that point. Figure 5 was generated by applying this kernel to the velocity field of a simulation of the interaction of a shock wave and turbulent flow field [Rotman 91]. The distribution function in this case calculated the height of a cylinder drawn in the direction of the average velocity. This height was then scaled by the magnitude of the vector. The Texl code for this filter is shown in Figure 6. Vector kernels that generate a more anisotropic reflection pattern are planned for future development.

The image in Figure 5 also shows the density and vorticity. Since both of these quantities had very dynamic ranges with only a few values in the range, a histogram equalization procedure was applied to the data image before applying the data mapping. These two images were then merged by showing only the right half of the density and the left half of the vorticity. A border was placed around the image and the color texture map and the vector bump texture map were passed to the shader. An animated sequence of this data set has been generated using these techniques.

Alternate representations

```
//
//  Pass a filter over the surface that deposits a cylindrical
//      brush in the direction of the vector field.
//
number vec_filt(
          surface vectors,  //  The data surface containing the vectors.
          number size,      //  The size of the vector kernel in pixels.
          number rad,       //  The radius of the cylinder in pixels.
          color a_color,    //  The overall color of the cylinder.
          surface out_surf  //  The resulting surface.
       )
{
   number xs = map_x_size( vectors );
   number ys = map_y_size( vectors );
   number ix, iy;
   number x, y;
   number temp;
   number length;
   number vx, vy;
   number x1, x2, y1, y2, s2;
   number min_length = 0.01;   //  Minimum vector length that is plotted.

   x1 = (size + (xs % size)) / 2.0;
   x2 = xs - size/2.0;
   y1 = (size + (ys % size)) / 2.0;
   y2 = ys - size/2.0;
   s2 = floor( (size-1)/2.0 );

//********************************************************
   //  Loop over the image, in strides of the kernel size.

//********************************************************
   for( iy = y1; iy < y2; (iy=iy+size) )
      if( (iy >= 0 ) && (iy < ys ) )
         for( ix = x1; ix < x2; (ix=ix+size) )
         {

//********************************************************
            //  Determine the vector and calculate its length.

//********************************************************
            vx = getcmp( vectors[iy,ix], 0 );
            vy = getcmp( vectors[iy,ix], 1 );
            length = sqrt( vx*vx + vy*vy );

//********************************************************
            //  If the vector's magnitude is greater than some
            //     tolerance, apply the kernel to this subarea.

//********************************************************
*
            if ( length > min_length )
            {
               vx = vx / length;
               vy = vy / length;
              for( y = (iy-s2); y <= (iy+s2); y++ )
                 for( x = (ix-s2); x <= (ix+s2); x++ )
                 {

//*************************************************
                     //  Calculate the distance the pixel is from the line
                     //     segment drawn in the direction of the vector
                     //     passing through the center pixel of the kernel.

//*************************************************
                     temp = vy * (x-ix) - vx * (y-iy);
                    temp = (rad*rad) - (temp*temp);
                       //****************************************
                     //  If the pixel lies within the cylinder, color it.
                       //*******************************************
                     if ( temp > 0.0 )
                      {
                       temp = sqrt( temp ) / rad;
                        out_surf[y, x] = length * temp * a_color;
                      }
                   }
                }
            }
         }
}
```

Figure 6. Texl code for the vector filter kernel.

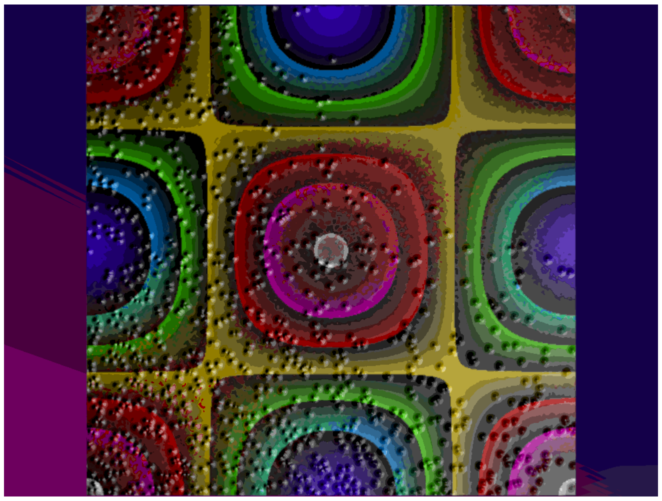

Figure 7. Representing scalar fields using random bumps and colored noise.

By using a stochastic noise function as described by [Perlin 85], one or more scalar functions can be represented based on different controls and uses of the noise. The most effective means for accomplishing this is to control the frequency of the noise function. Figure 7 illustrates three scalar functions. The most obvious function is the sin(x)sin(y) function represented by the blue to violet shade map. The function:

$$F(x,y) = a - x^2 - y^2$$

is represented by an increasing distribution of random bumps as we approach the lower right-hand corner. A third function:

$$F(x,y) = \begin{cases} \sin(x)\sin(y) & y < 1/4 \\ \sin(x+t)\sin(y+t) & y > 1/4 \end{cases}$$

is barely noticeable as a snow storm of noise where F(x,y) exceeds some threshold. While this last technique is not very noticeable with the shading and bump mapping, in an animated sequence, a very noticeable flickering will occur. While this is undesired in high quality animation, the amount of noise or flickering can be used to convey information, while at the same time, the natural filters of the eye/mind can ignore it.

Conclusions

We have shown the simplicity and effectiveness of using raster operations and texture mappings for the display of scientific information. Using textures to portray information allows the representation of several variables within a single image. However, the simple representations shown here are only the first steps to a more general and powerful representation of data sets. The research on glyphs could easily be incorporated into textures used in the rendering pipeline. Time varying textures such as the flickering of metallic paint and the boiling of water can also be used. The end goal will be to utilize the pattern recognition capabilities of the human mind, and to merge the disciplines of scientific visualization, image synthesis and image processing.

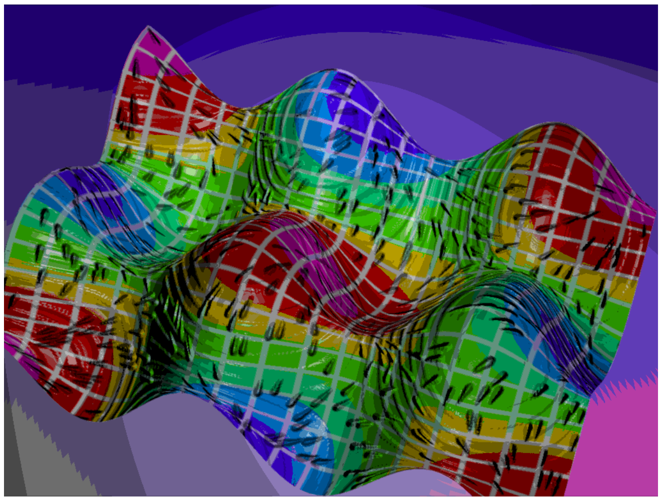

Figure 8. Textures generated in Figure 4 applied to a surface.

Finally, it must be remembered that the resulting representations of any of these techniques are images and hence can be used as texture maps (either color decals, bump maps, specularity maps or displacement maps). Mapping these textures onto surfaces such as surface plots are trivial (since there is a one to one correspondence between the surface domain and the texture domain). For textures generated in longitude/latitude space, these representation can again be easily mapped onto a sphere. This allows the flexible vector plots and isocontours to be applied to surfaces without having to rewrite new complex routines that specifically handle a vector plot on a surface, or produce a stack of plots in three-space, etc. Figure 8 is one such example, showing the image in Figure 4 mapped onto a surface plot instead of a flat plane.

Future Work

The research presented here is just the beginning of an extended program to examine the use of textures in both 2D and 3D. Our original intentions are to examine the use of textures in both 2D and 3D. Most of the techniques here can be quite easily extended for three-dimensional data sets, since the textures are generated procedurally. The use of anisotropic lighting models offers another avenue of research. Many of the techniques listed above need to be explored and tested on real scientific problems. For problems in which several variables must be understood at once, glyph-like textures may be explored.

Future work on the Texl system includes the ability to compile a Texl function to C with incremental compilation and dynamic loading. Optimization of the intermediate code representation (used by the interpreter) are also desired.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. We would like to thank Gene Cronshagen for generating Figure 5 and developing the histogram equalization techniques, and Doug Rotman for graciously providing the data for Figure 5. We would also like to thank Becky Springmeyer, Mark Wagner and Nelson Max for proofreading and useful comments.

References

[Cook 84] Cook, Robert L., *Shade Trees*, **Computer Graphics**, Vol. 18 No.3 1984 (SIGGRAPH 1984) pp. 223-231.

[Haeberli 90] Haberli, Paul, *Paint By Numbers: Abstract Image Representations*, **Computer Graphics**, Vol. 24 No. 4 1990 (SIGGRAPH 1990) pp. 207–214.

[Hanrahan 90a] Hanrahan, Pat and Paul Haberli, *Direct WYSIWYG Painting and Texturing on 3D Shapes*, **Computer Graphics**, Vol. 24 No. 4 1990 (SIGGRAPH 1990) pp. 215–223.

[Hanrahan 90b] Hanrahan, Pat and Jim Lawson, *A Language for Shading and Lighting Calculations*, **Computer Graphics**, Vol. 24 No. 4 1990 (SIGGRAPH 1990) pp. 289–298.

[Perlin 85] Perlin, Ken, *An Image Synthesizer*, **Computer Graphics**, Vol. 19 No. 3 1985 (SIGGRAPH 1985) pp. 287–296.

[Porter 84] Porter, Thomas and Tom Duff, *Compositing Digital Images*, **Computer Graphics**, Vol. 18 No. 3 1984 (SIGGRAPH 1984) pp. 253–259.

[Rotman 91] Rotman, Douglas, *Shock Wave Effects on a Turbulent Flow*, **Physics of Fluids, A,** Vol. 3 No. 7 1991 (July), pp. 1792–1806.

[Upstill 89] Upstill, Steve, **The RenderMan Companion**, Addison Wesley, Reading, Ma (1989).