Directly Rendering Spectral Elements Using Texture Shaders

Bernard Peng and Andrew Forsberg

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-02-17
2002

Directly Rendering Spectral Elements Using Texture Shaders

Bernard Peng and Andrew Forsberg

Brown University*

Figure 1: A zoomed-in comparison between the method described here (left image) and a quad drawn only using the corner data points (right image). The important idea is that more detailed information can be shown in the same quad. This left image is drawn with the following data:

427	282	640	632
347	226	542	543
307	200	492	494
325	214	512	510

Figure 2: This is the whole artery rendered with this technique. The element drawn in Figure 1 is taken from the highlighted element at the upper part of the artery.

Figure 3: Data locations on a 2nd order spectral element (left) and a finite element (right)

Abstract

We present a method to directly render data represented by high order polynomials on geometries using the Texture Shader in new graphics hardware. This work is motivated by the larger problem of understanding 3D fluid flow data sets and spectral elements [1], a data structure for representing simulated flow data. Our contributions include a way to interactively render surface data by evaluating and color mapping a high order Lagrange polynomial for every surface point in real-time. We have implemented this using OpenGL and NVIDIA's GeForce3 graphics cards.

Related Work Without texture maps, colors representing data values can only be specified at vertices of OpenGL geometries. Geometry tessellation is one approach to adding color (i.e., data) detail within an element. With texture maps, colors can be specified within elements, but require LOD techniques or choosing a texture map resolution and unfolding the textures across the geometry [2]. Much work has been done in these areas, but our approach differs in that neither texture maps representing specific color values or tessellation are used. Further, resources are proportional to the dimension and order of the polynomial, not the number of elements in a given problem.

Our Work Our method has two stages. In order to derive data values between quadrature points (see definitions), we use the Lagrange polynomial. Its formula for one variable is as follows:

$$p_n(x) = f_0\ell_0(x) + f_1\ell_1(x) + \ldots + f_{n-1}\ell_{n-1}(x) + f_n\ell_n(x) =$$

$$\begin{aligned} &= f_0\frac{(x-x_1)\cdots(x-x_n)}{(x_0-x_1)\cdots(x_0-x_n)}+ \\ &+ f_1\frac{(x-x_0)(x-x_2)\cdots(x-x_n)}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)}+\cdots \\ &\cdots+f_{n-1}\frac{(x-x_0)\cdots(x-x_{n-2})(x-x_n)}{(x_{n-1}-x_0)\cdots(x_{n-1}-x_{n-2})(x_{n-1}-x_n)}+ \\ &+ f_n\frac{(x-x_0)\cdots(x-x_{n-1})}{(x_n-x_0)\cdots(x_n-x_{n-1})} \end{aligned}$$

This can be extended to two and three variables. We begin by pre-calculating the coefficients for the polynomial (the ℓ -values). During runtime, we perform a dot product on these coefficients and the data values (the f-values).

Department of Computer Science, Brown University,
Providence, RI, 02912, {bpeng, asf}@cs.brown.edu

Calculating Lagrange Polynomial Coefficients After an element is mapped from world space into canonical space, its quadrature points are the same as every other elements. Since the Lagrange polynomial's coefficients depend on the location of the quadrature points, the coefficients are also the same for each element. These coefficients are pre-calculated, and then stored into a set of texture maps. Each pixel must have enough information to calculate the complete Lagrange polynomial for its corresponding point, so $n^2/3$ textures are needed to store the n^2 coefficients, each texture storing 3 values (in the red, green and blue channels). Because these RGB values are clamped to [0…1], there is one set of textures to store the negative values and one set of textures to store the positive values. The coefficients are also scaled down to fit them within the [-1...1] range.

Displaying the Data Using the Texture Shader we are able to perform a real-time dot product between the RGB values in one texture and texture coordinates we pass. We store the polynomial coefficients in a set of textures and pass data values as texture coordinates, so the resulting dot product is the solution to the Lagrange polynomial. Each pass only computes a dot product of 3 terms, so we additively blend each pass of textures that store positive coefficients, and then subtractively blend textures that store negative coefficients. The result is the sum of all the terms in the Lagrange polynomial, which gives us the derived data values between data points. The following is the pseudo code for the algorithm:

OpenGL Implementation Overview

```
Pre-Process
    load data values
    load pixel shader and texture shader
    pre-compute Lagrange polynomial coefficient textures

Render
    glEnable(GL_BLEND);
    glBlendEquation(GL_FUNC_ADD);
    glBlendFunc(GL_ONE, GL_ONE);

For every point:
    glTexCoord0(canonical coordinates);
    glTexCoord1(all pressure values);
    glVertex(world coordinates);

Texture Shader
    texture_2d();
    dot_product_2d_1of2(tex0);
    dot_product_2d_2of2(tex0);
```

Summary and Conclusion We have presented an approach to render high order data within a polygon without tessellation or generating specific textures for every element. We do so by using the Texture Shader to perform a real-time dot product, calculating the Lagrange polynomial to produce an image.

The advantage of our approach is that all of the data computed is accurately rendered. The real-time calculations of the Lagrange polynomial also allow us to visualize the data interactively. For example, the data can be scaled differently prior to computing the dot product to let a scientist look at a certain region in more detail. This is shown in Figures 1 and 2. Figure 2 is scaled so the scientist can see the data for the whole artery, while Figure 1 is scaled so the scientist can focus on the data within a specific element (the highlighted element on the upper part of the artery). If there are data for multiple time steps, they can easily all be loaded and animated to let the scientist see the change in data over time.

Another advantage is that the amount of texture memory required is fixed. It is only dependant on the order and dimension of the data given, not the geometric complexity. Thus rendering our artery dataset with 704 elements or our coil dataset with 3332 elements takes the same amount of texture memory.

This approach can also be extended to use 3D textures to draw the internal data within a spectral element. The scientist could use this tool to look at cross sections of data. We have rendered cross sections within a single element, but rendering a cut plane that spans multiple elements is future work.

Resources

Order	Textures(Passes)	Texture Memory (MB)
1	6	9
2	12	16
3	18	25
n	$2*(n+2)^2/3$	$2*(n+2)^2*256^2*8$ bytes

Performance

Order	FPS (artery)	FPS (coil)
1	24.39	6.45
2	14.92	3.79
3	10.75	N/A
Not using this method	25.64	7.57

Definitions

Spectral Elements - data structures that contain information about data at points within each geometric entity. Finite elements only hold information at the vertices. (see Figure 3)

Canonical and World Space - each element in world space maps to a position in canonical space, where calculations are done. An object in canonical coordinates lies within the unit cube. In this case, the Lagrange polynomial is calculated with points in canonical space.

Quadrature Points - for an n^{th} order solution, the n+2 positions along each axis at which the data values are known without evaluation.

Texture Shader - a programmable part of the hardware that takes texture coordinates and maps them to colors on a texture map.

References

[1] Karniadakis, G.E. and Sherwin, S. J., "Spectral/hp Element Methods for CFD", Oxford University Press, 1999.

[2] H. Battke, D. Stalling, and H. Hege, Fast Line Integral Convolution for Arbitrary Surfaces in 3D, Visualization and Mathematics (H. Hege and K. Polthier, eds.), Springer, 1997, pp. 181--195.