Dynamic Seismic Displays

Steven Lynch [1] and Larry Lines

ABSTRACT

Existing seismic displays, wiggle trace, variable density and terrain (SeisScape) displays are all limited because they are static. They provide a single view of a single seismic record or section. In fact, seismic processing is in general a static process because we apply a process and then view the result. We can view the data before the process or after the process, but using the traditional seismic display techniques we have no way to see anything in between.

In this paper we introduce a new type of seismic display, the dynamic display. A dynamic display is one which uses the programmability of modern graphics cards to perform limited seismic processing in real time. Dynamic displays can also produce a continuous sequence of displays. These displays can be configured to either gradually apply a seismic process or to gradually show how the process affects the data.

The advent of the graphic processing unit (GPU) has made it possible for us to pass multiple streams (versions) of seismic data to the graphics board. We can then use the GPU's programmable vertex processor to combine and manipulate the various streams.

Dynamic displays have the ability to fundamentally change the way that we interact with seismic data and with large and complex data sets in general.

INTRODUCTION

A typical seismic line contains an almost limitless amount of data that needs to be carefully processed and assembled before it becomes interpretable. The development of digital computers was a boon to seismic because it let us develop numerous techniques for doing this. Common techniques such as digital filtering, deconvolution and migration, which have so revolutionized the industry, were all made possible by the advent of the central processing unit (CPU) in the 60's and 70's. As much as the CPU revolutionized seismic processing, the advent of a new processing unit, the graphics processing unit or GPU, stands to make an even bigger impact on the interpretability of seismic sections.

Our current techniques for visualizing seismic displays, wiggle trace, variable density and terrain (SeisScape), displays, are all static. They let us view the results of applying a certain technique to our data but, in general, they do not let us apply processes themselves. Take, for example, the process of filtering a seismic section. We apply the filter in the CPU and then subsequently display the filtered data. We can display it before filtering and after filtering but we can't, using the existing display techniques, display anything in between. In our current seismic processing lexicon, there is no such thing as applying half a filter.

[1] Currently with Divestco Inc. and CREWES

Seismic processing functions are all or nothing and seismic displays are before and after. There is nothing in between and that is why we call the displays “static”. They provide a single look at a single section.

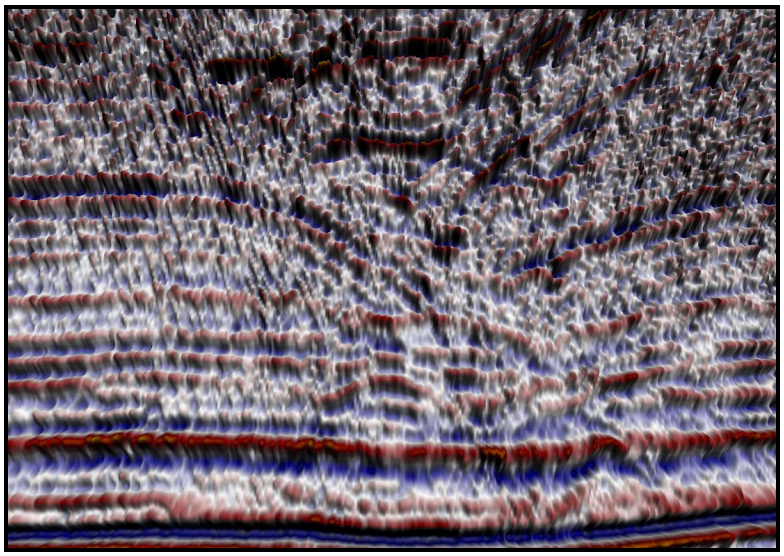

FIG. 1. Seismic section contaminated by steeply dipping noise.

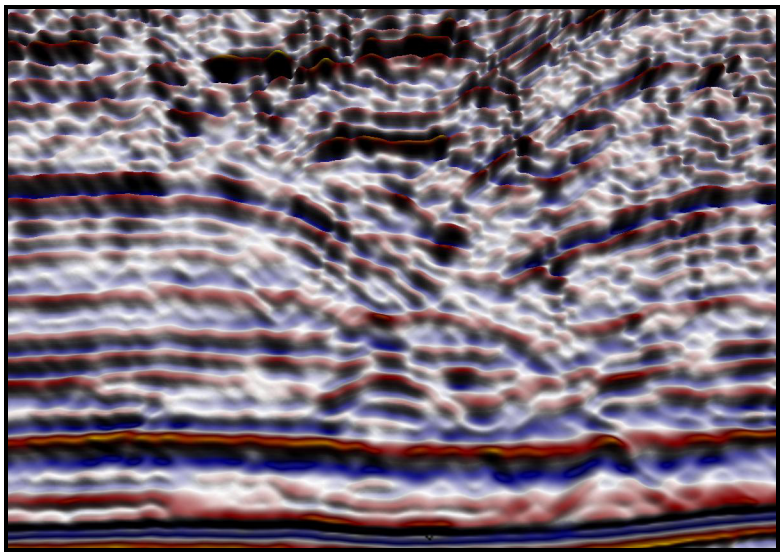

FIG. 2. The same section with the dips filtered out.

As an example, consider Figures 1 and 2 above. We start with the section shown in Figure 1, which is contaminated with steeply dipping noise. We then design and apply a filter that will remove it. The result, shown in Figure 2 is a section that has the offending dips and an unknown amount of desired signal removed.

When we interpret the data we can choose to use the original section, the one that is badly contaminated with noise or we can use the filtered section that has the noise and an unspecified amount of signal removed. But what we can’t do is see anything in between.

Both Figure 1 and Figure 2 are static displays because they show an image of the data at a particular point in the processing sequence.

But we can now do better than this. The development of the GPU means that we can replace these static displays with dynamic displays. That is, displays that let us apply processing steps in real time or combine multiple sections together on the fly. In this way the user can see exactly how the process has affected the data and they can choose how much of the process to apply in real time.

The graphic processing unit is the device that is responsible for the explosion in 3-D gaming technology and virtual reality. It is really nothing more than a very simple, very low-cost super computer. GPU's can't do a lot but what they can do, they can do a lot of and they can do it very quickly.

Unlike a CPU, the GPU is not a general-purpose computer. It is only optimized to perform specific functions. Most of these functions revolve around ways and means of realistically rendering complex surfaces in 3-D. This may, at first, seem to be of little use in processing and displaying seismic data. However, as we will show in the remainder of this paper, we can use the programmability of the GPU's to process and combine seismic sections in real time.

THE GPU

The primary goal of this paper is to show how we can break into the processing unit(s) on the graphic board and use them to manipulate and combine, in real time, various sets of seismic data. As a precursor to this it will be useful to describe the workings of the GPU and show how and where we can break into it.

Wiggle trace and Variable Density displays are built on a foundation of 2-D graphics. In this 2-D world, the majority of the display is created within the computers central processing unit (CPU) and the results are simply passed to the graphic card for presentation. This is fundamentally different from the world of 3-D graphics. In the 3-D world very little of the display is created in the CPU. Rather, the various display components are passed to the GPU, which subsequently is responsible for assembling the display. The CPU plays very little part in a 3-D display.

There are a number of reasons for off-loading a 3-D display from the CPU. Because of their general nature, CPU's must be able to handle a wide range of processing functions such as data input and output, communication etc. GPU's, however, are not general-purpose computers and only have a very limited set of capabilities. Because of this it is easier to optimize GPU's for specific tasks. GPU's are much less capable than CPU's but what they do, they do much faster.

The seismic terrain (SeisScape) display that is used throughout this paper is an example of a 3-D display. Unlike the wiggle trace or variable density display, it is produced almost entirely by the GPU. Vertex (sample) information is passed into the GPU, which subsequently lights, colors and renders the three-dimensional image of the section. It is possible to perform all of these functions in the CPU. However, unlike the

GPU, the CPU is not optimized for these types of processes and so a CPU based SeisScape display would render too slowly to be effective.

The Graphics Pipeline

The best way to think about a GPU is as a pipeline. One end of the pipeline is attached to the CPU and the other is attached to the display monitor. Various elements such as vertices (which in our world represent seismic samples) and textures (sample colors) are fed into one end of the pipeline from the CPU. They are operated on by the various stages in the pipeline, with the output from one stage being input to the next stage. The final output from the pipeline is a set of colored pixels, which are finally written to the display.

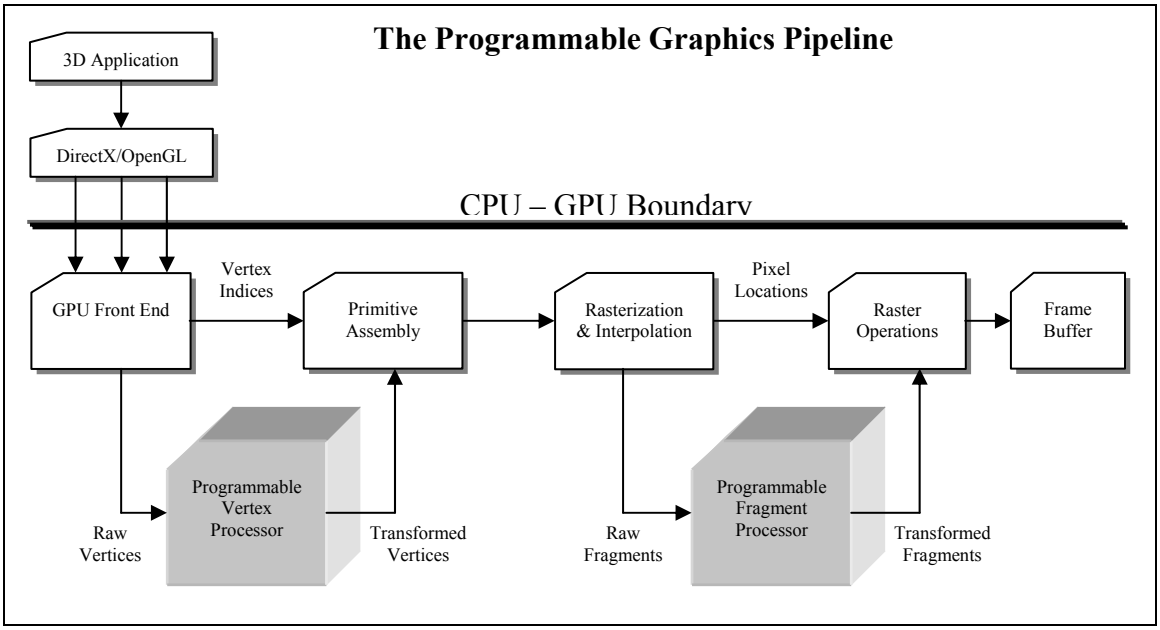

FIG. 3. The Graphic Pipeline

The pipeline usually has at least two input streams. The first is the set of untransformed (real world coordinates) vertices that represent the locations in real world coordinates of the various points in our 3-D model. In the case of seismic data, these vertices would be the seismic samples, i.e. the trace location, sample time and sample value.

The second input stream is a series of indices that tell the GPU how to assemble the 3-D surface from the input vertices. For the purpose of this paper we will consider that all 3-D objects are made up of a series of simple triangles, each of which has three vertices. The index stream is just a list of which vertices (samples) are to be used to assemble each triangle. This may seem like a lot of work considering that a seismic section, for example, may have millions of vertices (samples) but modern GPU's are optimized to assemble triangles this way.

Once the data has been fed to the GPU it goes through a series of stages. To start with, the vertex and index streams are separated. The vertex stream(s) are passed on to what is called the programmable vertex processor (PVP). This is the GPU unit that is responsible

for converting the untransformed real work vertex coordinates into screen coordinates. Once the PVP has done its work, its output is recombined with the index stream and the GPU then assembles the 3-D surface using the now transformed vertices and the original index stream. The output from this stage is a series of transformed and lit triangles.

The next stage in the pipeline is the Rasterization and Interpolation stage. At this stage the individual triangles are evaluated to see first if they should be displayed at all or whether they should be culled from subsequent processing. Triangles that aren't culled are then rasterized, which is the process of determining which pixels are covered by the triangle. There are two output streams for this stage. The first is a series of pixel locations, which are passed on directly to the final raster operation stage. The second is a series of corresponding fragments, which are branched off to the programmable fragment processor (PFP). Fragments are the basic building blocks for the final display. They contain a pixel location, depth, and various colors and texture coordinates. They are fed directly into the PFP for further processing.

The output from the PFP is finally recombined with the pixel stream and the recombined stream is fed into the final stages of the pipeline and then on to the GPU's frame buffer for display.

It is important to note that this is an evolving pipeline and Figure 3 only applies to fourth generation GPU's (NVidia GeForce FX series and ATI Radeon 9700). Third generation GPU's (NVidia GeForce TI series and ATI Radeon 8500) only have limited PVP's and their PFP's are extremely limited. Earlier GPU's (NVidia GeForce 2 and Radeon 7500 and earlier) do not have any programmable features at all.

As you might expect from their names, both the programmable vertex processor and the programmable fragment processor are accessible from outside of the graphic board. Prior to third generation GPU's they weren't. The vertex processor and the fragment processor were both configurable but they could only perform preset operations. Starting with the third generation it has been possible to write simple programs that are fed into the PVP and PFP from the calling application. The programs that we can write are still very simple, but as we will see they can be tremendously effective.

The remainder of this paper will deal with interactions with the programmable vertex processor, as that is where the majority of our real world processing will occur.

GPU Generations

An important thing to note for the remainder of this paper is that everything we will be talking about is only available on the latest GPU's. We are now at what is generally considered to be the fourth generation of GPU and the programs that we will be showing later only work on generations 3 and 4.

The first generation of GPU is generally considered to be the RIVA TNT and TNT2 as well as the ATI Rage and 3dfx's Voodoo3 cards. These cards, which became obsolete in 1999, are only capable of rasterizing pre-transformed vertices. That is, the CPU has to do the entire math that we now want to do on the GPU.

The second generation of GPU's (1999-2000) contains the NVidia GeForce 256 and GeForce 2, the ATI Radeon 7500 and the S3's Savage 3D. These GPU's are capable of performing the basic transformation and lighting of vertices thus offloading them from the CPU. They contain the ability to configure certain operations but in general they lack any form of real programmability.

The third generation GPU's, which include the NVidia GeForce 3 and GeForce Ti, and ATI's Radeon 8500, introduced true vertex programmability. However, the number of instructions is limited and whereas pixel shading is more configurable than early GPU's it is not really programmable.

The current fourth generation GPU's (late 2002 – present) included NVIDIA GeForce Fx cards and ATI's Radeon 9700 cards. These cards are the first to offload truly complex vertex transformations and fragment processing from the CPU. Although most of what we will be showing here will, with a few exceptions, run on 3rd generation GPU's, they run but they don't run very well. It is the fourth generation that has opened up the possibility of dynamic displays.

Programmable Vertex Processor

Although there are two places that we can break into the GPU, the PVP and the PFP, for the bulk of this paper we will focus on the programmable vertex processor. In the scheme of things, the PFP is just as important as the PVP. However it operates later in the pipeline, once the vertices have been processed and assembled and it is used primarily for creating special visual effects, which we will not go into here.

The programmable vertex processor is the engine that transforms the input vertices into screen coordinates and it is the first stage of the pipeline. Some of the operations that it can or should do are transforming vertex coordinates from real world to screen coordinates, calculating texture coordinates, calculating lighting and determining the vertex's color. It operates directly on the input raw data (seismic samples).

Remembering that in the seismic world, vertex equals sample, we can immediately see a problem with this pipeline. The pipeline outlined in the previous section is strictly sequential. That is, it operates on one vertex at a time. For seismic data this means that our vertex program can only operate on a single sample at a time. Depending on the GPU's capability, it may be parallelized for efficiency, but for the purposes of programming we can only process a single seismic sample at a time. This means that we can't do convolutions or FFT's in the GPU.

What we can do, however, is to pass multiple versions of a sample to the GPU. This means that we can combine various seismic sections together on the fly. It also means that we can apply any single sample technique, i.e. phase rotation, scaling etc. in the GPU. But we can't yet do true filtering.

For example, we could pass the GPU three streams of data. The first stream would be the trace (x) and sample (y) values for the vertex. The second stream could be the raw seismic amplitude and the third could be the corresponding quadrature trace values. With

these values we can then use very simple vertex programs to produce instantaneous amplitude displays, phase rotate the data, or both.

THE ANATOMY OF A BASIC SEISMIC DISPLAY TECHNIQUE

In the remainder of this paper we will introduce a small vertex program that will serve as an introductory Dynamic Seismic Display. The processes that we decided to include in the program are only representative of techniques that could be applied. We have chosen them because they illustrate the usefulness of the approach.

```
void DynamicTexture_V(
      in    float2      inputPosition   : POSITION,
      in    float2      inputTexCoord : TEXCOORD0,
      in    float inputzValue    : BLENDWEIGHT1,
      out   float4      outputPostion  : POSITION,
      out   float2      outputTexCoord0    :
TEXCOORD0)
{

      if( TimeVariantScaler > 0 )
            TimeVariantScaling(             inputPosition[1],
inputzValue );

      // Transform our position
      Clip( inputzValue );

      TextureScale( inputzValue, outputTexCoord0 );

      YAxisRotate( inputPosition.x, inputzValue );

      WorldTransform(     inputPosition,     inputzValue,
outputPostion );

      return ;
}
```

FIG. 4. The basic dynamic display program

```
void TextureColor (
      in float2 textureCoords    : TEXCOORD0,
      out   float4 diffuseColor   : COLOR0 )
{
      // get the diffuse texture color
      diffuseColor          =          tex2D(alphaSampler,
textureCoords);
}
```

FIG. 5. The basic fragment program

```
//
technique DynamicTexture
{
      pass P0
      {
            // shaders
            VertexShader      =      compile      vs_1_1
DynamicTexture_V();
            PixelShader             =    compile    ps_1_1
TextureColor();
      }
}
```

FIG. 6. The basic technique

At the right are three blocks of code. The first (Figure 4), *DynamicTexture_V* is the vertex program that will be loaded into the PVP. The second (Figure 5), *TextureColor*, is the fragment program that will be loaded into the PFP. The third, DynamicTexture, is called a technique. We will discuss techniques in the next section.

A Primer on Cg and HLSL

Many readers will already be familiar with the existing languages that we use to program the CPU. Languages such as Fortran and C have been around for some time and are well understood.

Mirroring the development of higher-level CPU languages, the original Shader programs were all written in assembly language and loaded onto the graphic board at execution time.

Over the past two years this approach has been replaced with what is called a High Level Shader Language developed in cooperation between Microsoft and NVidia. NVidia is the original developer and they called the new language Cg (g for Graphics).

Microsoft's implantation of this under DirectX is called HLSL (high-level-shader-language). Cg is a high level language and as its name suggests, it is comparable to the C language used to program CPU's.

Under Cg you don't load individual vertex and fragment programs, you load techniques. A technique, without going into too much detail, is responsible for compiling and loading the vertex and fragment programs for you. This may seem like overkill considering the simplistic way we are using it below. However, techniques can go far beyond what we are showing here and can in fact load different Shader programs for each rendering pass and can also set graphic states and options.

For a full explanation of the Cg language see the excellent reference "The Cg Tutorial by NVidia".

Referring to Figure 4, the vertex program – DynamicTexture_V, is doing the real work in this example and you can quickly see the relationship between Cg programs and C itself. Cg's syntax is very similar to C's but there is a significant difference in how things are passed in and out. Notice that there are five variables in the function, three marked with the "in" prefix and two with the "out" prefix. As you might expect, "in" simply means that variable is passed into the program from the pipeline and "out" means that it is passed on for further processing.

There are a couple of other new things. The float2 and float4 types do not exist in C. You can think of them as arrays of floats, which they are. They are just arranged a little differently internally so that they can be manipulated more efficiently by the GPU.

The reader will note that each variable is followed by an identifier (POSITION, TEXCOORD0 etc.). These identifiers are called semantics and they tell the GPU where to get the information from, in the case of "in" variables, and where to put them in the pipeline in the case of "out" variables. In the DynamicTexture_V function, for each vertex to be processed, the GPU will pass in the x, y position of the vertex (inputPosition), a set of texture coordinates (inputTexCoord) that in this case are not used and a further float value (inputzValue) which it will obtain from what are called the BlendWeights.

How all this information is assembled in the program and passed to the GPU initially is relatively straightforward and we won't go into it here. What is important is that this vertex program is expecting that the vertex z values will be from a separate stream than the x and y position values. This makes it much easier to work with when we come to use more involved vertex programs.

The Introductory Vertex Shader Program

We have chosen five processing steps for this introductory program:

1. Time Variant Scaling.
2. Amplitude Clipping.
3. Vertex Color Generation.
4. Rotation of the vertex Z value around the Y-axis.

5. Scaling the coordinates from real world values to screen values.

Each step is handled by a small function that is called from the Shader program. These functions are also written in Cg and are expanded inline when the Shader program is compiled.

Code Conventions

There are two types of code that we will show from here on, Shader code and calling app code. The Shader code is the code that is compiled in the GPU and will be shown using the following style;

```
void TimeVariantScaling( float time, inout float zValue )
{
    zValue *= pow( time / 1000, TimeVariantScaler );
}
```

The calling app code is the code that is used to configure the Shader's various input variables and data streams. It is compiled into the executable and will be shown in the following style:

```
BasicEffect.SetValue("TimeVariantScaler", TimeVariantScaler);
```

In this case we are setting the Shader internal value TimeVariantScaler with the value of the calling app's TimeVariantScaler. Note in most of what follows, for consistency, we use the same names in the Shader programs that we do in the calling app. However, this is not mandatory.

Code Efficiency

Vertex Shader functions are called at least once per vertex each time a scene is rendered. Considering that there may be millions of vertices (samples) in a seismic section and that we want to render the display in excess of 15 frames per second means that the our vertex programs will be executed hundreds of millions of times per second. As a result, when programming them you have to keep in mind that every step you add is adding hundreds of millions of execution instructions per second. This means that vertex programs, to be efficient, must be very small and compact.

In fact there are limits to the number of instructions that can be compiled for a Shader program. DirectX 8.1 programs have a limit of 128 instructions whereas DirectX9 have 256. As a consequence, programming vertex shades is very much like stepping back 20 years to the time when you had to worry about every instruction and every byte of memory. Still, as you will see, even in this limited (but rapidly expanding) environment we can still achieve some remarkable results.

Step1: Time Variant Scaling

The first function that we chose to include in my basic effect is Time Variant Scaling. We included it because many of the test data sets that we have are uncorrected shot records and we wanted to have the ability to correct them on the fly. We also wanted to test the efficiency of a function that applied a different correction for each sample.

Considering what we said in the previous section on efficiency this may seem like a waste of time. There aren't a lot of situations where time variant scaling is required, however we found that the comparison:

```
if( TimeVariantScaler > 0 )
```

which is called for every vertex, did not affect the overall execution speed and so we left it in.

```
BasicEffects.SetValue("TimeVariantScaler",
TimeVariantScaler);

float TimeVariantScaler;
void TimeVariantScaling( float time, inout float zValue )
{
      zValue *= pow( time / 1000, TimeVariantScaler );
}
```

FIG. 7. Time Variant Scaling Function

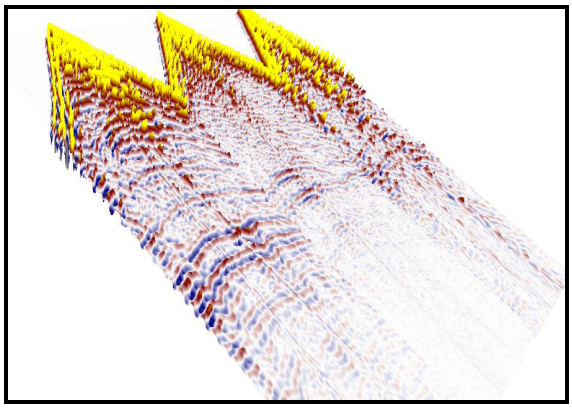

FIG. 8. Raw shots records with no correction applied.

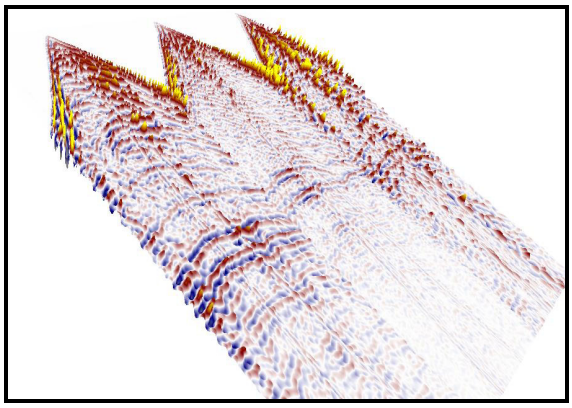

FIG. 9. Raw shot records with a TVS of 1.0 applied.

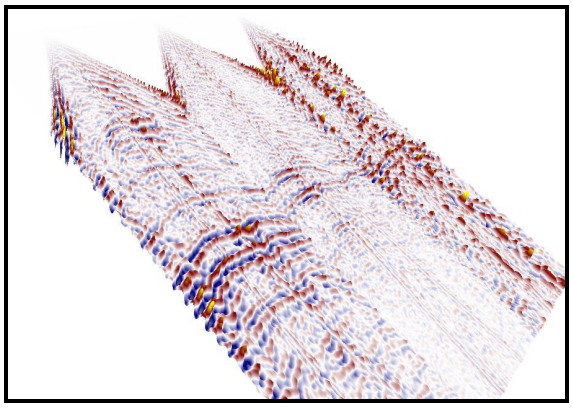

FIG. 10. Raw shot records with a TVS of 2.0 applied.

Figure 8 is the code that executes the scalar, and as you would expect it is very simple. The only thing to note is the "inout" keyword that precedes the "float zValue". This just means that the variable is passed into the function and that its manipulated value should be passed back out.

Figures 9-11 show the raw records, the records with a T Function of 1.0 and a 2.0 respectively. The applied values 1.0 and 2.0 do not represent anything fixed in the program. In reality, the scalar is tied to a slider control, which manipulates the value in real time. The effect of doing this on the data is quite dramatic so much so that we tend to use it to better balance even stacked and already corrected sections. Unfortunately, as

with most of the examples to follow, the dramatic effect of animating the scalar in real time is impossible to show on paper.

Step 2: Clipping

Clipping is a useful technique for enhancing weak events and so we chose to include it in my basic display program. One of the major problems with seismic data is that quite often, the events we are interested in have weak amplitudes when compared to the rest of the section. Having such low amplitudes make them very hard to identify and follow on traditional displays.

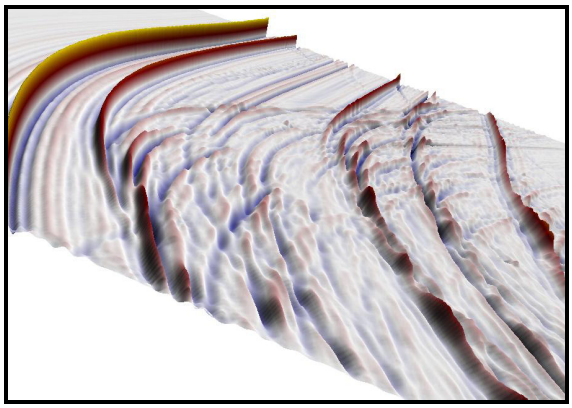

FIG. 11. Unclipped section with scaling of 1.0

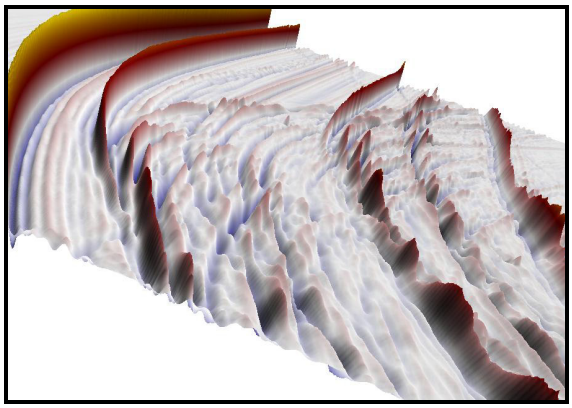

FIG. 12.Unclipped section with scaling of 3.0

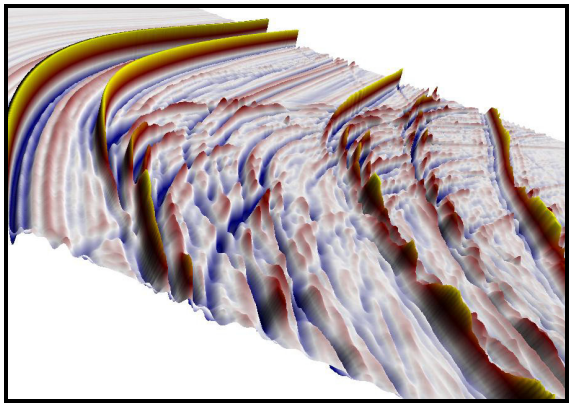

FIG. 13.Clipped section with scaling of 1.0

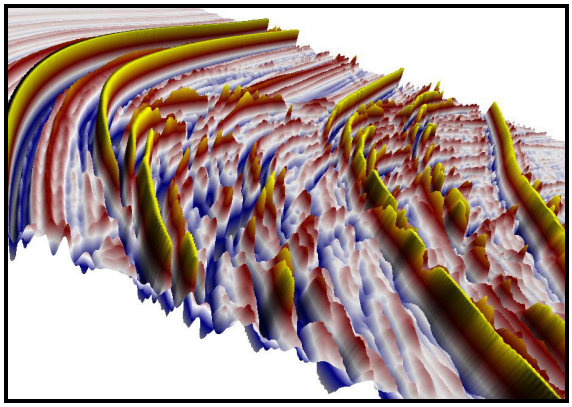

FIG. 14. Clipped section with scaling of 3.0

Because of their three-dimensional nature, SeisScape displays get around this problem to a large extent but as you can see from Figure 11 we could do better. In this example, the middle of the section has a number of weak events whereas there are numerous strong events that tend to dominate the display.

The easiest way to boost the amplitude of the weak events is to increase the amplitude scaling which we have done in Figure 12. However, this is not the ultimate solution because as you can see the strong events are also boosted and so the display becomes even harder to use. Note also that the scaled section shown in Figure 12 has the same

color mapping as the original – Figure 11, and as a result the display in not greatly improved.

SeisScape displays are inherently three-dimensional but there is no real reason that they have to be displayed as surfaces. In fact, it is just as easy and efficient to display each trace as a wiggle (Figure 18). We can also, if we want, display both the wiggles and the surface itself (Figure 19), to produce what is very similar to a normal variable density wiggle trace overlay.

```
BasicEffects.SetValue("PeakValue", PeakValue );
BasicEffects.SetValue("ClipValue", PeakAmp.Value );

float PeakValue;
float ClipValue;

void Clip( inout float z1 )
{
	if( z1 > ClipValue )
		z1 = ClipValue;

	if( z1 < -ClipValue )
		z1 = -ClipValue;

	z1 = z1 * PeakValue / ClipValue;
}
```

FIG. 15. Code for the PVP clipping function.

The solution to this is to use a real time clipping function (Figure 15). In this function we pass in the peak value of the seismic data (this is always passed in as it is required by the Texture Scaling function) and a clipping value. The function itself is self-explanatory.

Applying this function produces much better results as is shown by Figures 13 and 14. In this case, the low amplitude events have been scaled in both amplitude and color and the high amplitude events are clipped so that they don't distort our view.

In practice we find that we use this function more than any other because it dramatically improves the display.

Step 3: Vertex Color Generation

To color the seismic mesh we create a one-dimensional texture that contains all of the colors in the palette, in sequence. The code in Figure 16 calculates the appropriate texture coordinate for a sample (values between 0.0 and 1.0) and returns it.

```
// Code for two-sided data (i.e. seismic)
void TextureScale( in float inputzValue, out float2
outputTexCoord )
{
      inputzValue = (0.5 + (0.5 * inputzValue /
PeakValue));
      outputTexCoord[0] = 0;
      outputTexCoord[1] = inputzValue;
}

// Code for one-sided data (i.e. dt curves, rhob curves)
void TextureScaleOneSided( in float inputzValue, out
float2
outputTexCoord )
{
      inputzValue = inputzValue / PeakValue;
      outputTexCoord[0] = 0;
      outputTexCoord[1] = inputzValue;
                              }
```

FIG. 16. Code to generate the texture coordinates.

There are two functions, one for two-sided data, that is, data such as seismic that is expected to oscillate around 0, and one-sided data, such as interpolated log data, that is not.

Step 4: Y Axis Rotation

The Y-Axis rotation came about because of my desire to produce a display that was similar to the standard variable density/wiggle trace overlay that is common in many interpretation programs.

```
BasicEffects.SetValue( "CosVal", (float)Math.Cos( AxisRotation.Radians) );
BasicEffects.SetValue( "SinVal", (float)Math.Sin( YAxisRotation.Radians) );
      float yscaler = (float)(HeightScaler * TraceSpacing / PeakValue);
            BasicEffects.SetValue( "XScale", yscaler );

                  // input variables
                    float CosVal;
                     float SinVal;
                     float XScale;

            void YAxisRotate( inout float x, inout float z )
                              {
                  // Transform our position
            x           = x + z * SinVal * XScale;
                  z           = z * CosVal;
                              }
```

FIG. 17. Code to rotate the z value around the Y-Axis

SeisScape displays are inherently three-dimensional, but there is no real reason that they have to be displayed as surfaces. In fact, it is just as easy and efficient to display each trace as a wiggle (Figure 18). We can also, if we want, display both the wiggles and the surface itself (Figure 19), to produce what is very similar to a normal variable density wiggle trace overlay.

If we look at Figure 19 we see that what we have produced a display that is very similar to the standard w.t./v.d. display. The only difference is that z value (or trace excursion), which is normally in the plain of the section, is rotated by 90 degrees around the y (trace) axis. To produce a traditional w.t./v.d. display all we have to do is to apply a phase rotation on the fly, see Figure 17 for the Shader function that does it.

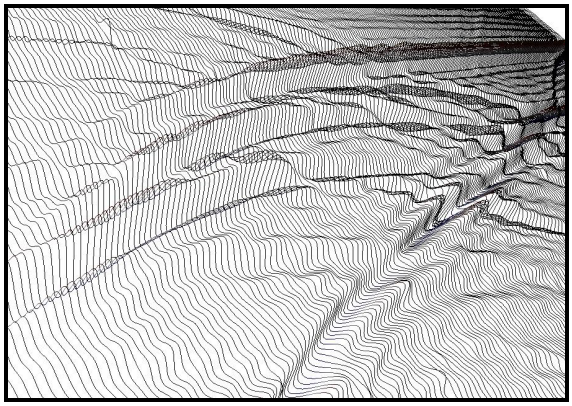

FIG. 18. Vertical Wiggle Traces

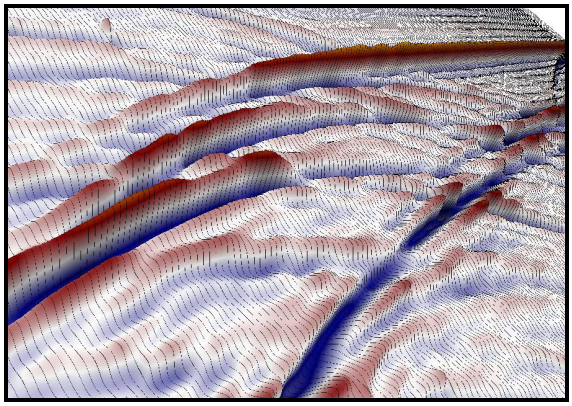

FIG. 19. Vertical Wiggle Trace Overlay

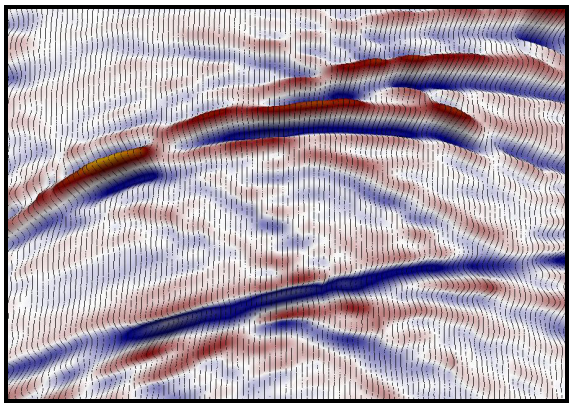

FIG. 20. The same display as Figure 19 but looking vertically down.

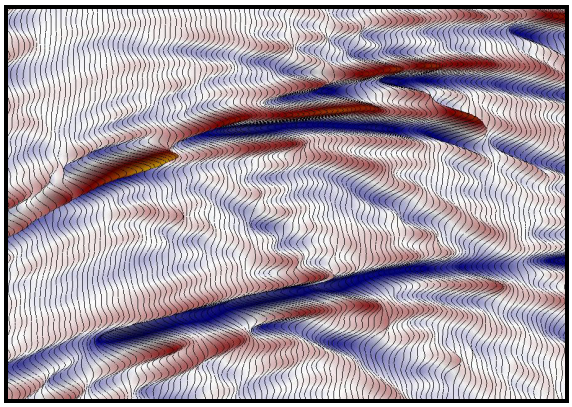

FIG. 21. “Normal” Wiggle Trace Overlay

The inputs to this function are the sine and cosine of the desired phase rotation. We could calculate these values on the fly by simply passing in rotation value but then the calculation would be required for every vertex that we process. Sending the sine and cosine directly avoids this overhead.

There is one other parameter that we pass in, the XScale. This is a value that adjusts the rotated values for the effect of the amplitude scalar that we normally apply to the z values.

As you can see from Figure 21, which shows the display rotated by 90 degrees, the result is very much like the standard variable density/wiggle trace overlay. We should point out that using this code, any phase rotation is possible and only values around 90 and 270 degrees will appear flat. We haven't found any use for phase rotations other than 90 even though the effect is quite interesting and we may eventually take out the user controllable value and replace it with a flat/elevation toggle. But we will leave it in for now because who is to say how some enterprising young geophysicist will make use of it.

Step 5: World Transformation

The final step in the basic program is to scale the x, y, z coordinates, which at this point are in real world coordinates, into screen coordinates. We won't go into any detail here because this simply involves the application of a transformation matrix and the subject is covered in just about any text on 3D graphics such as "Tricks of 3D Game Programming Gurus" by LaMothe.

CONCLUSIONS

In this paper we have shown how it is possible to use a computer's graphic processing unit to perform limited seismic processing in real time. The GPU is still early in its development and the programs that can be written for it are very simple. However, as we have shown, it is possible to perform simple processing functions such as scaling and clipping on the graphic board, in real time.

Although not presented here for brevity, it is also possible to pass multiple versions of seismic samples to the GPU and perform other more interesting tasks such as phase rotation, real time attribute analysis and morphing between sections. These will be presented in subsequent papers.

Future developments of the GPU could let us perform more complex tasks such as convolutions and FFT's in real time.

REFERENCES

Akenine-Möller, Tomas and Eric Haines, "Real-Time Rendering", A. K.Peters
LaMothe, André. "Tricks of the 3D Game Programming Gurus", SAMS, 2001
Lynch, Steven, "Composite Density Displays", CREWES Report, **15,** 2003