

GPU-BASED DISPLACEMENT MAPPING FOR SIMULATION APPLICATIONS

Antoine Azar
B.Eng. Student
Department of Computer Engineering
École Polytechnique de Montréal
antoine.azar@polymtl.ca

Abstract

I introduce in this paper a solution for a completely GPU-based displacement mapping applied in a scientific simulation context. More particularly, I provide a high-level algorithm to compute element displacements within a mesh, with vector values applied at each node of the mesh. Several possibilities involving the vertex and the fragment shader are explored. While an ideal solution would use the vertex shader, current hardware and API limitations force us to use the fragment shader, in a multiple-pass rendering algorithm using 2D and 3D textures to store and modify the simulation data.

Keywords: Displacement mapping, numerical simulations, GPU, multipass rendering, vertex shaders, fragment shaders.

1. Introduction

Many scientific experiments, simulations and measurements involve the displacement calculation of a series of punctual elements (particles), given certain parameters like stress, temperature, pressure, etc. It is often the case that these parameters will be entered in a simulation software, which will then output a resulting mesh with a solution at each node. It becomes then practical to study displacements within that mesh. Examples of applications include the simulation of an anatomical deformable model under stress, like breast compression which is used in cancer detection, the flow of water particles in a turbine, or the deformation study of a lattice in a bridge.

While a finer mesh and a large number of elements displaced will provide more precise results, the processing load on the CPU will become increasingly important. However, with the modern advance of GPUs (Graphics Processing Units) on graphics cards, more and more computations can be performed onboard, while leaving the CPU free to work on other processes.

It is the goal of this research to port as much workload as possible from the CPU to the GPU while computing and displaying the displacements of each element in the mesh.

2. Numeric simulations

The PLOT3D format is very popular in numeric simulations. The format supports two types of mesh grids: structured and unstructured. In the case of a structured grid, the dimensions of the grid are well defined, and the nodes are spaced at regular intervals. In an unstructured grid, the nodes can be anywhere in space, in any arrangement. This research will focus on structured grids in order to keep the focus on the real problem at hand, but switching to an unstructured grid should not pose any major additional problems.

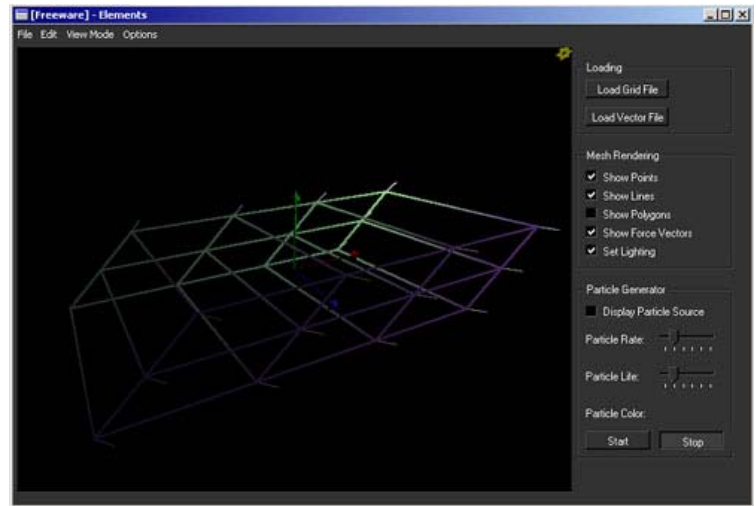


Figure 1 - Example of a simple structured grid in a custom written software

3. Anatomy of a GPU

Traditionally, graphics cards incorporated fixed function hardware, which was capable of accelerating specific API functions, like OpenGL Gouraud Shading. Visual effects and computation possibilities were thus limited to the API's functionalities.

With the advent of programmable units, called GPUs or VPU's, it becomes possible to a developer to write his own code to process vertices sent to the graphics cards and to perform pixel processing. For example, the programmer can write his own rendering algorithm instead of using OpenGL's default shading equations.

In order to perform this, two major units have been incorporated in the graphics chip: vertex shaders and fragment shaders (also called respectively programmable vertex processors and programmable fragment processors). Figure 2 shows a simplified diagram of a GPU.

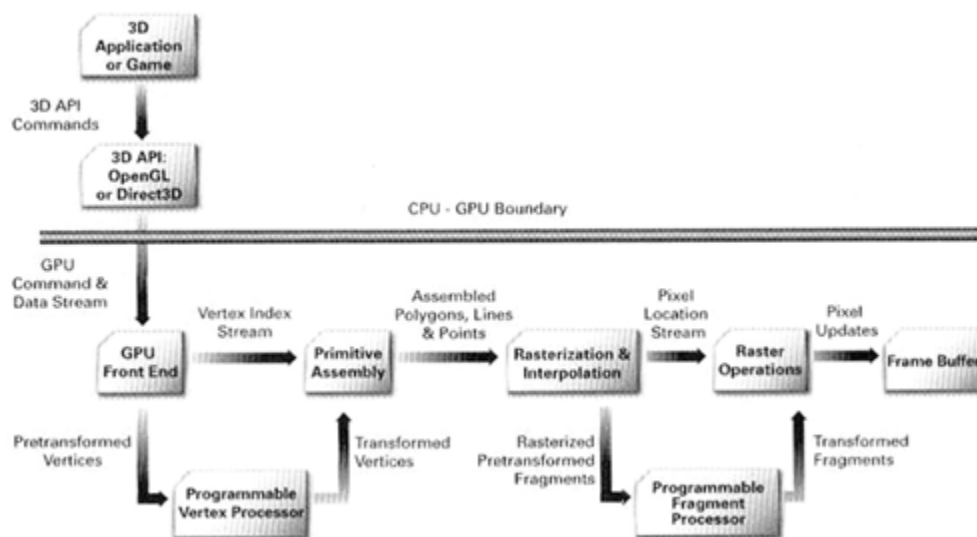


Figure 2 - Simplified diagram of a GPU [1]

As seen on figure 2, the vertex shader is the first element in the processing chain. It takes as an input vertex parameters, performs all necessary transformations, and outputs assembled primitives.

These primitives are rasterized (transformed into fragments, or “potential pixels”), and are sent to the fragment shader. The fragment shader executes its code, applies all textures, and outputs the transformed fragments. The last step of the rendering is determining which fragments are really pixels of the image, and how exactly to draw each pixel.

As we can see here, each shader has access to different data and each is capable of different processing tasks, and of accessing different memory spaces. We will study the possibilities offered by each one.

4. Simulation Optimization

It is necessary to identify our critical operations in the simulation process, and to determine if optimizations via GPU computing is possible. Two tasks stand out as the most costly and the best suited for our work: the interpolation of the values of a particle’s surrounding nodes, and the computing of the particle’s next position. These two tasks consist of simple mathematic operations, but the number of time they have to be computed in a complex simulation can stress any CPU.

5. Computation via the Vertex Shader

Each particle being traditionally represented by one or more vertices, it makes sense to use the vertex shader to compute element displacements and speeds.

Mesh Modeling via a 3D Texture

The mesh is a series of nodes with a vector applied at each node. It is possible, using a texture, to represent up to a four dimensional vector in a single texel of an RGBA texture, by simply projecting vector components onto color components. Each texel would then represent a node. If the node values are one-dimensional, such as temperature, it is then possible to pack several scalars into one texel, thus reducing storing size, and also adding the possibility of SIMD (Single Instruction Multiple Data) parallel computing. In the present case, we will be projecting a node’s 3D vector onto the RGB components of each texel

As a vertex would be sent to the vertex shader, it would only be necessary to perform one texture lookup in the 3D texture at the vertex’s position to know exactly the value applied at this point. GPUs are highly optimized for processing and accessing textures, and (bi/tri)linear interpolation would be automatically and efficiently done.

Unfortunately, current vertex shaders do not have access to texture data (except for texture coordinates). This limitation should be removed with the next graphics card generation, when the VS 3.0 standard will be defined. The *OpenGL Shading Language*

seems to already allow this feature via the extension ARB_Vertex_Shader, but it is a software emulation that is currently being used.

Mesh Modeling via GPU Constants

It is possible to use memory reserved to constants on the GPU to represent the current vertex's surrounding nodes. However, current GPUs have relatively few constant registers, and this number varies greatly from graphics card to graphics card.

It is thus impossible to represent the entire mesh in this memory space. We would then have to implement a multi-zone segmentation algorithm that would determine on-the-fly in which zone the current vertex is, and load in the constants memory the values at these nodes. The GPU would then have to compute the interpolation of these values and calculate the next vertex position.

This approach is theoretically possible but does not seem to present any real advantage over a pure CPU solution. The CPU would still be required to determine the correct zone, and we would incur a performance hit by constantly loading new values in the GPU.

Other Considerations

Although the 3D texture solution seems optimal, another problem arises from the use of the vertex shader: its output is only sent to the fragment shader for rendering. Thus, the element's new position would never be updated in the main memory, and each iteration would be similar to the last one (switching from frame 0 to frame 1). There is currently work being done to allow vertex shaders to directly modify the data, but the current graphics cards and APIs do not support it.

6. Computation via the Fragment Shader

Due to the limitations of the vertex shader, we have to turn to the fragment shader for a solution to this problem. This shader, however, has been designed to process fragments and has no knowledge of a vertex's position or attributes. However, if we consider multiple-pass solutions (rendering several times to obtain the final result) and imagine new uses for the given resources, new possibilities appear.

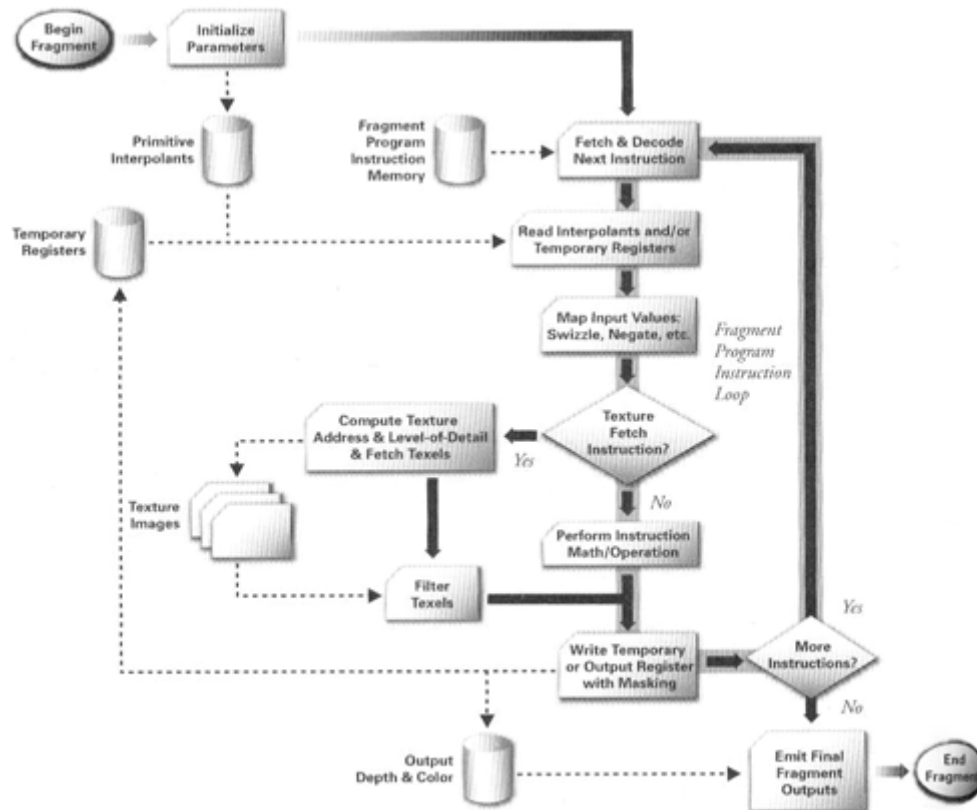


Figure 3 – Flow Chart of a Fragment Shader [1]

Data Modeling via a 3D Texture and Multiple 2D Textures

As in the first vertex shader solution, we model the mesh as a 3D texture by projecting vector components onto color components. As shown on Figure 3, the fragment shader has direct access to texture information, and we can thus easily determine the vector values to apply to any particle position. In order to perform the displacement simulation, we also have to give the shader access to the particle's position and speed, which we can encode in 2D textures (1D textures could also be used). Here is the high-level algorithm of this solution:

- Read simulation file (ex: Plot3D)
- Encode the mesh in a 3D texture by projecting vector components onto color components
- Read in element attributes (initial position, speed, etc)
- Encode each element attribute in a 1D or 2D texture
- For each new frame:
 - o Load the fragment shader
 - o Render in the back-buffer any one of the attribute texture onto a viewport-aligned polygon*
 - o In the fragment shader, for the 0th to nth element:
 - Read in the elements' attributes by performing texture lookups at the exact position of each element in the texture
 - Read the mesh's interpolated vector values at the element's position by performing a lookup in the 3D texture
 - Compute the element's new attributes
 - Render back into the attribute textures using Multiple Render Targets (MRTs)
 - o Convert the position texture to a vertex array
 - o Unload the fragment shader
 - o Render the vertex array to the back-buffer
 - o Swap front and back buffers

*In order to perform the computation on each element, we need to make sure the rendered polygon respects a perfect texel-to-pixel alignment. That way, the fragment shader will be called once for each texel.

Another problem arises: we now need to convert a texture information into vertex positions. A few solutions are possible:

SuperBuffers

SuperBuffers are memory spaces allocated on the graphics card which allow for various uses. Typically, a memory space is reserved for a specific task: a framebuffer, a texture, a vertex array, etc. SuperBuffers remove this limitation. We would use this technique to output the new element positions by rendering the position texture in a SuperBuffer, which would then be interpreted as a vertex array. The second rendering pass of our algorithm would then simply display the vertex array, and the elements would appear at their updated positions. Unfortunately, SuperBuffers have not yet been implemented by graphics cards vendors, but should become available sometime in the end of 2004.

CPU Readback

Another possibility is to simply perform a readback on the position texture via calls like `glReadPixels` (in OpenGL). Each value can then be read individually and interpreted as a position for a vertex. This however implies CPU processing, and stresses the AGP bus, as the data must travel back and forth from the graphics card to the PC's RAM.

API Extension

A solution half-way between the two previous ones would be to use API extensions that allow the transfer of texture data to a vertex array directly on the graphics card, without using the PC's main memory. Nvidia, for example, already offers extensions that allow this, like `NV_Pixel_Data_Range` et `NV_Vertex_Array_Range`. This solution doesn't offer the flexibility of SuperBuffers, but should still offer a much better performance than the simple CPU readback.

7. Conclusion

After studying the various options available from current hardware, drivers and APIs, we find that the optimal solution to perform a GPU-based displacement simulation is to use the fragment shader. All necessary information needs to be encoded in textures, so the fragment shader can read in the data, and render back the solutions. A second pass is then necessary to display the elements at their new positions. This non-trivial operation would be best performed by SuperBuffers when they become available, but needs for now to be done via a vendor-specific extension or by a CPU readback.

If the vertex shaders are improved in the future to access textures and to output values to a memory location, they would greatly simplify the task. There would be no need for texture-encoding of the elements' attributes, and only one pass would be required per frame.

Book References

- [1] Fernando R., Kilgard M., The CG Tutorial : *The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, USA, February 2003, 336 pages.
- [2] Woo M., Neider J., Davis T., Shreiner D., *OpenGL Programming Guide*, 3rd edition, Addison-Wesley, USA, October 1999, 730 pages.
- [3] Watt A., Policarpo F., *The Computer Image*, Addison-Wesley, USA, 1999, 751 pages.

Article References

- [4] Goodnight N., Woolley C., Lewin G., Luebke D., Humphreys G., *A Multigrid Solver for Boundary Value Problems using Graphics Hardware*, Proceedings of Graphics Hardware 2003, 2003, 4 pages.
- [5] Percy James, *OpenGL Extensions*, ATI Research, Siggraph 2003, 42 pages.
- [6] Green Simon, *Stupid OpenGL Shader Tricks*, nVidia Corporation, Game Developers Conference 2003, 29 pages.
- [7] Moule Kevin, *Making Good On The Real-Time Promise*, University of Waterloo, 2001, 29 pages.
- [8] Harris M., Coombe G., Scheuermann T., Lastra A., *Physically-Based Visual Simulation on Graphics Hardware*, University of North Carolina, USA, 2002, 11 pages.

Internet References

Harris Mark, *Dynamic Texturing*, nVidia Corporation.
<http://developer.nvidia.com/attach/1250>

CodeSampler.com, *OpenGL (1.2-1.5) Code Samples*
<http://www.codesampler.com/oglsrc.htm>