Jumping Into The Deep End
An OOP Approach To GPU Game Architecture

[Paper Submittal: vgsp_0036]

Jason S. Hardman
DarkWynter Studios

Games and Learning Lab & Regional Visualization Center
University of North Carolina at Charlotte
Jason.S.Hardman@gmail.com

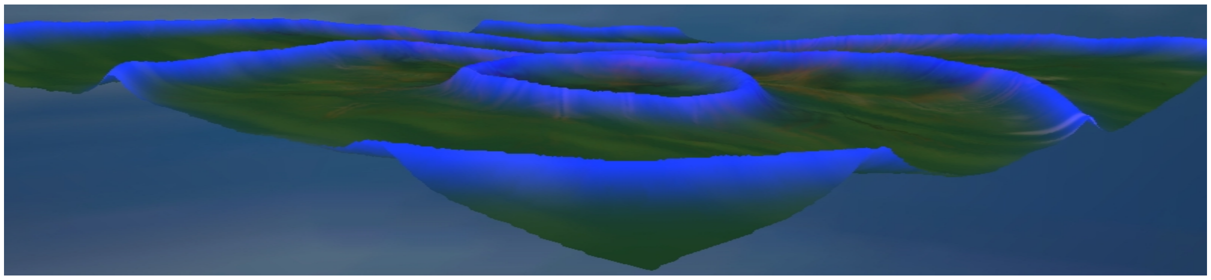

Figure 1) Use of a wave form algorithm in the Vertex Shader

Abstract

The current generation of gaming platforms (Xbox360, Play Station 3) includes phenomenally fast Graphics Processing Units (GPUs) that allow the developer to program directly on the GPU. This technology is called Shader Programming.

Far from being a simple "airbrushing" technique, it has been shown that Shaders have all the necessary components for doing General Purpose computations on the GPU (GP-GPU). Given the phenomenal speed of the GPU compared to the CPU and its reputation for beating Moore's law (3), this technology opens a doorway into a whole new paradigm of game development...if we can figure out how to use it.

In common practice, all this blindingly fast computational power remains unharnessed in video games today. This is best exemplified by the short falling of Play Station 3 (PS3) games, which occur despite the impressive statistics of the Cell processor.

What follows is an outline of a general programming structure compatible with ATI, nVidia, and XBox GPUs. Research suggests that the same techniques can be applied to the PS3, though development in this area is just being undertaken by the author.

Keywords: GPGPU, Shading, Application Architecture

1 Introduction

The most common reason for the lack of GPGPU computing in video games is the expense of moving information across the CPU/GPU bridge. It is often suggested that this is a major limiting factor and that GPU algorithms will be hindered until the transfer speed increases.

We propose that the problem is not bridge speed, but a hesitation of the industry to jump into the deep end of the pool. If enough computation is pushed over the GPU, we do not need to pull these values back to the CPU. For most application developers, this will require a firm commitment to building a robust architecture for handling physics, as well as other modifiable variables, in shader code. It also requires a new way of thinking in terms of exactly how a GPU is supposed to be used.

2 Why The Fuss?

On a practical level, GPU development is hard. Even the best tools are still in their infancy, and debugging especially is tedious. There are no text based outputs, no break statements, and rather cryptic error messages.

As such, the best current approach seems to be to develop, debug, and stabilize a CPU physics model first. While this will generally increase the development time of the project, it has benefits beyond simplifying development. In particular, this code can be left in the application to allow support for lower model hardware by inserting a few switch statements based on hardware capabilities.

3 CPU Code Organization

From a survey of examples, tutorials, and books on the subject, most shader development is currently being done on a case-by-case basis. Individualized algorithms are applied to each class of objects. This is very much against the principles of code reuse and modularity.

We propose a more general architecture for organizing GPU resources in a manner that promotes function reuse across multiple objects.

One of the keys to constructing a general architecture for the GPU is the initial organization of CPU information. By letting each object handle its own draw and update functions, the task of organizing becomes easier.

Information and methods to be ported to the GPU, such as an object's physics properties and methods, should be collected and organized using Object Oriented Program (OOP) encapsulation techniques. Information passed to these methods can then be treated like signals and passed to the GPU in a robustly defined manner.

In our game, EleMental, we have set forth the purpose of avoiding smoke and mirrors and creating a fully interactive environment. This includes the manipulation of several thousand particles representing base elements such as earth, wind, and water.

We started this process by defining what we are calling a Mass object. It contains all physics properties of an object such as position, mass, scale, velocity, and adhesion. By creating a Mass object for every object in the environment, we maintain consistency of format, which will be beneficial to us as we begin converting these values into a GPU friendly data format. The only physics methods that cannot be encapsulated in this way are those related to collision detection, which inherently must be able to compare multiple objects at a time.

Figure 2) Elemental by DarkWynter
A game based on real world physics and particle interaction

Once this organization is done, the process or porting to the GPU becomes simpler. All that is left is to convert the Mass encapsulated physics functions into Shader code.

Once this port is done, the collision functions can handle passing values to the shader, which handles updating the physics values on the GPU.

4 GPU Data Organization

CPU to GPU data types come in two flavors:

Global Variables
Uniforms are standard read/write variables that are set by the CPU, and can be accessed by either the Vertex or Fragment Processors. They retain their values until the CPU modifies them (7).

Uniforms can also be used as control signals to a larger architecture. This type of usage is addressed later in this paper.

Texture Data Objects
The use of textures as a medium for data can be complicated and should generally be ignored until an application begins bottlenecking at the CPU/GPU bridge.

Except for RenderToTexture techniques, textures can be thought of as read-only data. Textures are a good way to pass large sets of relatively static data to the GPU. For organizational purposes it is suggested that each object be assigned to a row and each property of the object to a column or vise-versa.

Stream Structures
Stream values originate from the CPU with variables like position, texture coordinates, and normals, and are passed from vertex to fragment shader using varying variables. These variables are interpolated by the rasterizer as they are passed from the Vertex to the Fragment processor. These values eventually come to rest in the frame buffer or frame buffer object depending on the programmer's decision.

It is very useful to organize these variables into separate structures for vertex input, vertex output, fragment input, and fragment output. Larger applications may also want to include intermediate structures for organizing properties internal to the vertex or fragment shader.

By mapping these structures to the underlying semantics provided by the hardware (e.g. position, textureCoordinateN, colorN), we decrease the likelihood of "crossing wires" as we move information from the input semantic to the output semantic. This translation is necessary because different semantics are used for the vertex input, vertex output, fragment input, and fragment output.

For example, the Vertex Shader Input semantics include variables for tangent and binormal. These values must be mapped to texCoord variables if they are to be sent to the Fragment Shader. The number of variables that can be interpolated is very limited so a consolidated set of variables must be chosen wisely.

We recommend including normals, binormals, tangents, screen vectors, and light vectors in this translation process. These variables, along with the object space, world space, and view space matrices, provide a mathematical basis from which other values can be derived.

5 GPU Instruction Sequencing

It is recommended that both the Input and Output structs for Vertex and Fragment shaders should be passed to their respective sub functions. This ensures that each function has access to both the incoming variables and any previously calculated output variables. The sequence of function calls is very important, as it must ensure that values calculated in one function do not clobber previously calculated values.

Both vertex and fragment shaders should begin by establishing a mathematical basis using the incoming stream structures. This makes these values available to any other functions that require them and helps to maintain code cleanliness and organization.

Vertex effects, in particular, must be carefully sequenced because the modification of a position or texture coordinate by one function can greatly affect another. In general, position-modifying functions should be saved until last to guard against errors and clobbered values.

Fragment functions deal more with color and, thus, are a bit safer to work with. It should be noted that each function should check for previously calculated output colors and use blending to modify the value. In particular, it is suggested that lighting calculations be saved until after color modification functions have completed. This will ensure that the final output color is correctly lit.

6 Control Signals

Simple CPU signals like the current position can be passed to the GPU using Uniform variables. For simple applications or applications with a small number of values to pass, this can be a very effective approach.

Uniform signals apply to the object in its entirety. World location and orientation parameters for a dynamic object are a good example of proper usage. Other recommended uses include incrementors for controlling animations and parameters to scale effects based on temperature, intensity, or velocity.

For applications where the overhead of transferring information to the GPU creates a bottleneck, the use of Texture Data Objects and RenderToTexture technology can provide a reasonable solution. This requires either a second update pass, or the ability to render to multiple targets, one being the frame buffer and the others being the object textures.

7 Instructions, Clock Cycle and RenderToTexture

RenderToTexture from the GPGPU perspective is often thought of as a data output mechanism providing feedback to the CPU.

The basic idea is to take data from the texture into the shader, run computations on it, and render it onto the same texture using a unit-quad. To ensure data integrity, a one-to-one texel-to-pixel ratio must be kept. This can be done by setting the viewport size to that of the texture and rendering in orthographic mode. Multiple textures can be updated per pass by applying multiple textures to the quad at render time and adjusting the viewport accordingly.

From an engineering perspective it should be noted that this technology marks the end of a clock cycle for the GPU processor. Thinking in these terms leads us to the natural question of how much time is necessary to complete a complex instruction. Since shader code runs at different frequencies dependent upon the length and computational complexity of the code, we must also address the question of whether a RISC (Reduced Instruction) or CISC (Complex Instruction) style is preferable for general-purpose GPU architecture.

From examining publicly available code, we have found that most shader programmers have applied the CISC methodology, creating long and complex shader programs to attach to their objects. This approach has the natural disadvantage of reducing the re-usability of this code in other objects due to the specialization of its instruction.

A medium ground is found in modularizing a general shader into functional units that can be called from the main program. This is an effective strategy for increasing the portability and reusability of shader code, and is the strategy that we have adopted in EleMental. It should be noted that this is not a very effective strategy for GPGPU heavy applications, however, because it is difficult to modify the sequence in which function calls are made while ensuring that variable clobbering does not occur.

Another approach makes the observation that shading languages are still very similar to the assembly languages they are built on. By using a RISC approach to shader creation, we can create a small and complete set of independent shader "instructions", which run much faster than the traditionally long and complex shaders.

By using RenderToTexture at the end of each shader call, it is possible to batch process a large number of shader "instructions" in-between graphical frames. This appears to be a very effective strategy when using VertexBufferObjects to drive the process, as the overhead of sending vertexes, normals, and coordinates across the expensive CPU/GPU bridge is avoided.

Processing values this way requires extensive use of the discard keyword in the shader to avoid overwriting values that are not applicable to a given operation.

8 GPU Memory Management

For applications with extremely volatile data, memory management techniques must be applied to the GPU textures to reduce fragmentation, which is caused by objects being added and removed from the textures.
.
A traditional approach is to pull this information back to the CPU, reorganize the data, and push it back to the GPU. This is extremely slow. To maximize efficiency, we want to reorganize the texture data on the GPU itself. A solution to this problem can be found in the vertex processor.

We must first note that a there is a relationship between the Texture Coordinate (input location of texture data) and the Vertex Coordinate (output location of the texture data). By rearranging the vertexes of a texture, we can effectively move the data to another location on the texture.

We have tested this theory using single points for each texture location and have found a 10x slow down vs. using a four point quad. This loss shows a trade off between the precision with which data can be reorganized and computational speed.A recommended strategy that balances this trade off is to use a spreadsheet format (see:Texture Data Objects) to organize multiple objects onto a single texture. This approach allows us to create, relocate, or remove a row or column at a time using a single quad for each row or column. It also allows us to modify or pull back rows or columns of values to the CPU instead of pulling the entire texture.

Reorganization (defragmentation) can be done by changing the location of the corresponding vertices (output location) of rows or columns that are currently being used to ones that are not in use.

9 Conclusion

While facing a variety of unique and complicated challenges, we feel that the use of the GPU as a common processor for applications has a tremendous future in the graphics world and in gaming especially. The challenge is up to us, however, to rethink everything about how games are made and our programming techniques. This natural progression will lead us to a whole new way of programming games where parallel processing of object physics can eliminate many of the traditional smoke and mirror techniques. Moving away from these traditions and on to more realistic technology will open the door for truly immersive and interactive environments and lead us to the point of rivaling the very world we strive to emulate.

References

1. 3Dlabs Inc. Ltd. *Ogl2particle.h*. 2-22-05. 3-23-07. ttp://scottdouglas.net/ projects/glsl/fire/ogl2particle.h
2. Kilgrad M. J. *Graphics Hardware Functionality For Geometric Computations With OpenGL*. NVIDIA Corporation, 2002.

3. Lin, M. C., and Manocha, D. *Interactive Geometric and Scientific Computations Using Graphics Hardware*. 2003. SIGGRAPH 2003 Course Notes, vol. 11. ACM SIGGRAPH, July, ch. 1--6.

4. Lindholm, E., Kilgard, M. J., And Moreton, H. *A User-Programmable Vertex Engine*. Proc. SIGGRAPH 2001 (July 2001), 149–158.

5. Peercy, Mark S., Marc Olano, John Airey , and P. Jeffery Ungar. *Interactive Multi-Pass Programmable Shading*. Proceedings of SIGGRAPH 2000 (New Orleans, Louisiana, July 23-28, 2000). In Computer Graphics, Annual Conference Series, ACM SIGGRAPH, 2000.

6. Pharr, Matt (ed). GPU Gems2. R. Fernando, Ed. Addison Wesley, Reading, MA. 2005.

7. Rost, Randi J. OpenGL Shading Language, Second Edition. Addison-Wesley Professional, January 25, 2006.

8. Rudolf, Florian. *GLSL – An Introduction*. Neon Helium Productions. Article 21. 3-1-06. 3-23-07. http://nehe.gamedev.net/data/ articles/article.asp?article=21

9. Wloka, M. M. *Implementing a GPU-Efficient FFT*. In ACM SIGGRAPH 2003. Course Notes, pages 132-137. ACM SIGGRAPH, August 2003.