Debugging GPU Programs

Krzysztof Niski
Johns Hopkins University

Abstract

A method for debugging arbitrary vertex and fragment programs written in Cg will be presented in this paper. In this method the user is allowed to specify a variable to debug, as well as operations to perform on the variable to be debugged. Subsequently the user has the ability to step through the vertex and fragment programs and watch the changes in the values of the selected variable. The method also allows further visualizations of the collected data, such as line integral convolution or symbolic representations of the data.

The method uses bison to perform a breakdown of the original Cg program, and then creates new versions of the program that terminate when the user specified criteria is reached, and returns the value of the variable of interest after performing some user specified action on that variable.

1. Introduction

Vertex and fragment programs are pieces of code that are used by graphics hardware to process vertices and rasterize individual pixels. Rather then performing these calculations in traditional serial order, the graphics hardware designers have moved towards parallelism in video cards, so that each operation of a vertex or fragment program is performed simultaneously on multiple data, effectively using a SIMD architecture.

These programs have become ubiquitous in modern rendering, performing a multitude of tasks, ranging from simple per fragment lighting operations, down to performing advanced mathematical and physical simulations as well as surface deformation and animation. The fragment and vertex program instruction sets, although limited, give the programmer the ability to perform a plethora of tasks, including branching and looping in the latest incarnation of graphics processors. As the latest graphics cards become more powerful and more general in their abilities, the complexity of the programs increases as well.

The programs can be written in a variety of languages, starting with OpenGL ARB, an assembly-like language, to high-level languages such as glSL and Cg that follow a C/C++ syntax, which is compiled into code native to the graphics card. The Cg language, which is the language that will be used in this debugger, is a C style compiler that converts the Cg code into a form compatible with the graphics card, such as ARBFP, NV pixel shader or NV fragment program. The advantage of this language is its cross platform capabilities, both in its support for various graphics card architectures, including ATI and NVIDIA products, as well as support for Linux, Mac OSX and Windows. It is worth noting that although Cg was chosen for this paper, this method can be easily modified for any of the other languages, such as glSL or HLSL.

While the various implementations of the vertex and fragment programming languages give ample feedback as to the correctness of the syntax of the program, they are unable to verify its proper function. Because the operations are performed on hardware to which there is no low level interface, and the memory of which is not accessible, the program cannot be disassembled and debugged in a traditional manner. The only thing that can be received from a fragment program is the final result, in RGBA form, which is written to the frame buffer.

This means that there are only two ways to debug a fragment program: Either perform the rendering and program evaluation in software, or break down the vertex and fragment programs, and force them to return the variable of interest back to the frame buffer to allow inspection.
In this paper a method will presented for debugging fragment programs using the second approach, which gives a more accurate debugging environment. Because it uses the actual graphics hardware, it will be capable of providing true results from the programs.

2. Previous Work

Debugging of programs is an area as old as programming itself. When developing software it is always beneficial to have the ability to peer inside the workings of the application in order to determine where the faults lie.

The simplest form of debugging is performed by simply analyzing the program line by line, and comparing inputs to outputs. While this approach may work, it is extremely inefficient and prone to human error. Print statements can be used to aid in this process, but this method is still extremely time consuming even for moderately complicated applications.

Since most applications are fairly complex, a better method of debugging applications was required. This was provided by text based applications such as gdb [Stallman 1989] or dbx [Linton 1990]. These debugging environments are familiar to most programmers working in *Nix systems, and provide an interactive environment which allows breakpoints, stack traces, variable value lookups and a host of other features. Graphical interfaces to these debuggers have been since developed, such as DDD [Zeller and Lütkehaus 1996], and incorporated into development suites such as Microsoft Visual Studio or Apple XCode.

However, since both the fragment and vertex programs are more akin to parallel applications then serial ones, these tools and their methodology is insufficient. To overcome the difficulty of debugging such programs multiple debuggers have been designed. Applications such as pdbx or TotalView are capable of debugging distributed and multithread processes by extending the traditional serial processing debugging paradigm. While functional, these applications still have problems with large scale parallel applications, where the data and process interaction becomes more complex.

For massive parallel applications, environments such as Prism [Sistare 1992] or MPPE allow mapping of data from multiple cpus to graphical representations on a single screen, giving the operator a chance to monitor the data flowing in and out of processors.

The method of debugging fragment and vertex programs is most akin to the massive parallel application debugging. Each pixel in the output framebuffer is essentially the end result of a parallel processor. Just like the massive parallel visualizer, we want to be able to view the data inside a closed vertex and fragment processor.

There are already several different systems capable of debugging graphics hardware. They can be divided into two opposing camps. On ones side we have applications which use the graphics hardware to debug the vertex and fragment programs by various means. On the opposite side, we have graphics card emulators which perform the debugging and rendering in software.

The simplest method of debugging a fragment program is to simply manually rewrite it. While tedious, this works. Several applications exist to simplify and automate this process, such as Shadesmith, which can step through a fragment program written in ARBFP and can output the selected variable to the framebuffer and perform numerical lookups of values. Another application, imdebug, is of displaying the framebuffer after a fragment program is executed in order to numerically analyze the data. Apple includes a OpenGL Shader Builder with their developer suite which can be used to create and debug fragment and vertex programs in a closed environment.

In the software rasterization area Microsoft provides the Shader Debugger Tool, which uses software rendering, providing access to both fragment and vertex program data. Similarly Mesa3D, a software OpenGL implementation for the X window system, could be used obtain debug data from inside OpenGL.

While all of these methods are capable of providing some information and outputs from a fragment program, they are incapable of visualizing the vertex program data, and also are unable to analyze a higher level vetex/fragment programing language in hardware. Additionally the hardware based methods do not add support for obtaining true floating point values from the vertex and fragment programs, severely limiting the usefulness of the application.

3. Cg Program analysis

The first stage of performing the debugging is the interception of the Cg fragment program before it is given to the graphics card. To make the debugging truly non-intrusive requires intercepting the Cg and OpenGL calls that are performed when the Cg program is loaded. This, however, is outside the scope of this paper, and a

simpler implementation which subtly alters the source code can also be used.

One of the key aspects to this problem is the ability to take an arbitrary vertex or fragment program written in Cg and break it down into individual lines and statements. This is achieved using semantic analysis, which is performed by a combination of bison and flex. The original syntax definition file for Cg was obtained from NVIDIA, and was subsequently stripped and rewritten, as this application does not require the amount of information provided by NVIDIA's flex/bison parser. The parser implemented builds up a tree structure of the program, making it possible to later traverse this tree, and reconstruct a partial version of the program, which will be used to return the value of the variable the user is interested in.

The parser must not only break down the program into lines, which is trivially performed by searching for line break or semicolons. The result of such a analysis would be unusable due to the fact that Cg programs, unlike ARB assembly, follow a C style syntax. This means that branching, looping and context is supported, and violating any of those constraints would render the resulting program uncompilable. The parser is therefore capable of identifying such features in the program, and storing them in a tree structure, which can be subsequently traversed.

This is accomplished by having the bison parser call a tree structure function whenever certain elements in the code are located. For example if an *if* statement is found in the code, a tree branch is created of type if, whose text entry will contain the if statement, and whose children will be the code to be performed inside the *if* statement. The tree nodes also contain other information, such as whether they are variable declarations, and if so the type of variable, as well as line information to match the user defined variable to a line. Once the entire program is stored in this tree it can be traversed and used to reconstruct either the original program or a modified version of it.

4. Program reconstruction

Once the program is analyzed and the data structure is created, a new program can be rewritten from the original vertex or fragment program.
When the user specifies a variable to debug, the original program is searched for the definition of that variable in order to determine the variable data type. Once the data type is known, a custom terminator block is written which will take the data the user wishes to return and cast it into the output type from the main program. The terminator can also take several operator hints, allowing it to scale the output value, or return the value if it lies within a specified range.

To create a program the tree data structure is traversed and checked for the terminating condition, which is the line number of the variable we want to return to the frame buffer. When this condition is met, the customized terminator is appended to the recreated program. In order to maintain the syntactical structure of the program the terminator may be modified depending on the location of the variable, for example if it is located in a loop or a conditional statement.

To perform the reconstruction we have to include all of the necessary structure and function definitions, which have to be placed in the correct spots in the program. The functions may also be eventually inlined and edited, so it is necessary to store as much information as possible from the original program during the parsing process.

4.1 Common Problems

Although the process of rewriting the program seems relatively simple, there are numerous issues encountered when attempting to piece together a new program after rewriting parts of it.

The main quandary is that of syntax and code cohesion created by the limitations of the hardware, and therefore the programing language. For example we need to consider what to do when the variable to be inspected is inside a conditional block. Due to fragment and vertex program limitations conditional returns are impossible; we must therefore rewrite the *if* statement in a way that returns a value in both true and false conditions, and we also need to discern which output the user asked for. This is achieved by returning the variable in the case the user asked for, and returning a dummy value otherwise. This problem is further exacerbated by the possibility of the *if* statement occurring inside loops, functions or within another conditional statement. The task of rewriting such a nested statement becomes more involved, as it not only has to follow C style syntax, but also has to obey the constraints of the Cg language.

Another involved problem is the one of functions. While functions are very useful at eliminating some of the code from a program, they also mask some possible operations on the variable of interest. In order to alleviate this problem we inline the functions when necessary, which gives us the ability to break inside it only at the instance the user specified rather then every time the function is called.

4.2 Rewriting programs

This process performs the basic reconstruction of an arbitrary program. The reconstruction for a vertex and fragment program is not, however, identical.

Fragment Programs

The simpler case of the two is returning data from the fragment program. This already has a instinctive mapping to pixels in the frame buffer, since the result of a functioning fragment program will return color values to the frame buffer. The nature of the fragment program, therefore, makes examining variables inside it relatively simple, as each fragment being debugged has a one to one mapping with a pixel in the frame buffer. The program needs to be rewritten so that the value to be inspected is set as the output color and sent to the frame buffer. As discussed previously, a terminating statement is appended to the program in order to return the correct variable to the frame buffer.

The debugger also has a couple of other possible modes for rewriting the fragment program. One of them is the case of returning data from a vertex program. In this case, the original program is discarded, and a new program is made from scratch. This fragment program simply returns the variable debugged inside the vertex program, which was passed to it via one of the texture coordinate inputs.

Another, more complicated, mode is depth peeling. To perform this we need to add a section of code to the beginning of the program that will receive a depth texture from GL, receive the window position components of the fragment, and the depth value of the current pixel. This code segment will then perform an early fragment termination if the depth of the pixel is greater then the depth in the texture. The main problem is in adding the inputs to the program without disturbing the inputs the program already has. When Cg compiles a program, it removes dead code very aggressively, and can remove inputs to the program, as well as renumber the inputs that are kept in the compiled code. As a result, the numbering of the inputs frequently changes when a program is terminated early due to debugging, or if the number of inputs is changed or the inputs are reorganized inside the Cg program. To bypass this problem the parameter calls by Cg have to be intercepted and redirected in the new program. A similar process is performed for textures, except that we must bind textures to different texture units in order for the Cg program to access the correct texture. Because Cg usually handles textures on its own, the debugger needs to determine which texture unit Cg has assigned the depth texture, and bind the depth texture to that texture unit.

Vertex Programs

Rewriting vertex programs is more complicated than fragment programs due to the non-intuitive way vertices are mapped to the fragment program. While it is possible, and sometimes useful, to simply perform the full transformation pass of the vertex program and simply return the value of the vertex variable in the fragment program, it is not very instructive. The three main problems with this approach are the fact that GL will interpolate the values of the variable across triangles, and if rendering triangles, they will obscure the fragments, and therefore vertices behind them. While these problems can be overcome with point rendering, the most important issue is not: even if we know one vertex to be supplying incorrect data, we have no way of knowing which vertex it is. Because the vertices are rendered as an object, identifying a single vertex and viewing its true floating point value is impossible.

To overcome this we rewrite the final frame buffer coordinates which the vertex program returns to the fragment program. The coordinates are rewritten in a way that lines them up in a row and column manner in the frame buffer. This is achieved by passing in a additional vertex attribute when the vertex is issued to OpenGL. Using one of the unused vertex attributes we pass a vertex id and window size to the vertex program by modifying the original program. Inside the vertex program itself we add a code section which will find the variable the user has defined as the output position, and change it to the new position obtained from the windows size and vertex id.

5. Data Visualization

Visualizing the data is relatively trivial for a fragment program, but becomes somewhat complicated for the vertex program output due to the reordering of vertex data.

The result of a fragment program debugging pass is a scene in which the objects rendered that use the debugged fragment program are colored based on the value of the variable being inspected.

For example if the fragment program performs manipulation on normals, and something goes amiss, we can visualize the normals on the model by displaying them as RGB values corresponding to the XYZ values of the normal, as shown in figure 1.

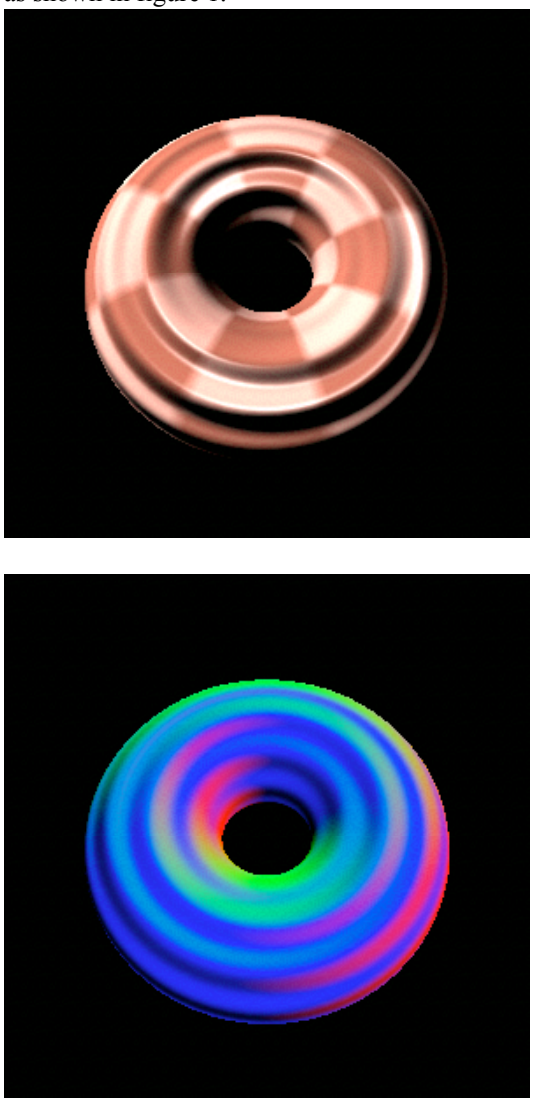

Figure 1. Bump mapped torus (top) with the corresponding normals obtained from the debugger (bottom)

This is the first step in visualizing the data from the debugger. The data is converted into a meaningful format for the display, which means it has to be in the range 0-1, as this is the limit on the RGB output of the frame buffer. Because we are trying to visualize the data and look for patterns and areas where the fragment program fails, rather then analyze the actual numerical data, this approach is sufficient, providing the data can be scaled so that it can be visualized.

There are several visualization options that can be performed inside the fragment program. Currently, they are normalize, scale, length and range. Each of these visualization methods allows us to view the data in a different way. The most important part is to scale the value so it is in the range 0-1, so that it can be visualized in the frame buffer. This can be accomplished by either the normalize or the scale call. We can also look at the data without scaling by using the range operand, which will color the fragments outside that range a specific color, leaving the fragments inside the range in their true value. Finally we can also visualize the absolute value of the vector, ie. its length, so the output of the fragment becomes a single scalar representing the variable.

It is also possible to visualize the data by performing operations on the resulting data read from the fragment program. When the data is read from the graphics card, it is simply an array of floating point values, arranged in quads representing the RGB and A components from the frame buffer. The depth buffer is also read back from the graphics card in order to perform data visualization techniques.
In order to fully visualize the normal vectors we must also abandon the standard frame buffer in favor of a pbuffer, which is an offscreen rendering target capable of storing values in their full float range, rather then forcing them to be clamped to 8 bit values in the range 0..1.

After reading the float values from the pbuffer to a buffer in main memory, various methods of visualization can be applied. The display of this data varies greatly upon the type of data stored in the buffer. If the data to be displayed are normals at each fragment, many methods exist which will transform the normal field into a understandable form. One of the more meaningful methods for normal field visualization is the use of symbols as shown in figure 2. By drawing arrows or lines from a fragment in the direction of its normal we

can quickly determine the direction of the normal vector.

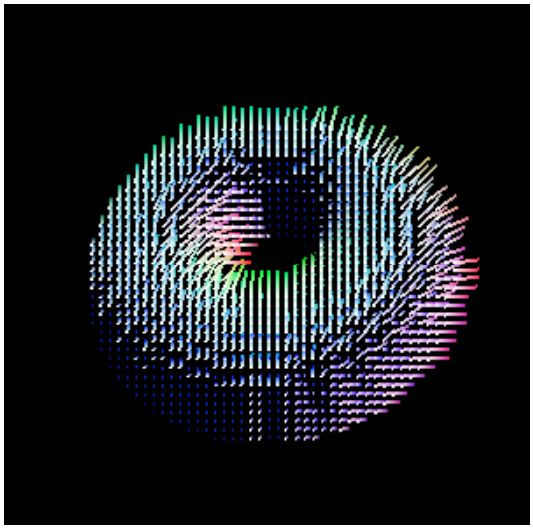

Figure 2. Displaying fragment normals using symbols. The lines represent the direction of the normal

While this method has the advantage of clearly showing the normals, it cannot display the normal of every fragment, because the result would be too cluttered. As an alternative we can also use line integral convolution (LIC) methods, which can show a flow representation of the normal field.

Visualizing vertex program data becomes more complicated. The easiest way to display the output is to simply transfer the reordered vertex data straight to the frame buffer or a pbuffer. This can be quickly visualized, but does not present the data in an intuitive way the way rendering fragment program data does.

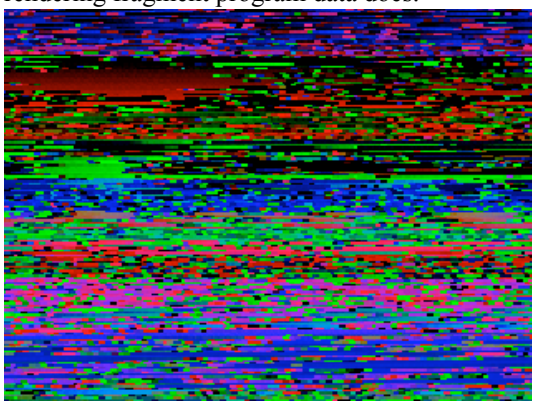

Figure 3. Frame buffer output from a rewritten vertex program. Each pixel corresponds to a normal from a vertex

Although the data is conceptually hard to visualize, it can be matched per vertex to the vertices being sent out by the video card over the course of several rendering passes. For example we can render the vertex IDs along with the vertex normals and vertex positions, and then compare the vertex normal and position to the issued vertex using the vertex ID. This of course cannot be performed directly, but can be done using additional software. To display the data directly we can simply re-render the output from such a program in order to make sure all of the vertices and normals are transformed properly, as shown in figure 4.

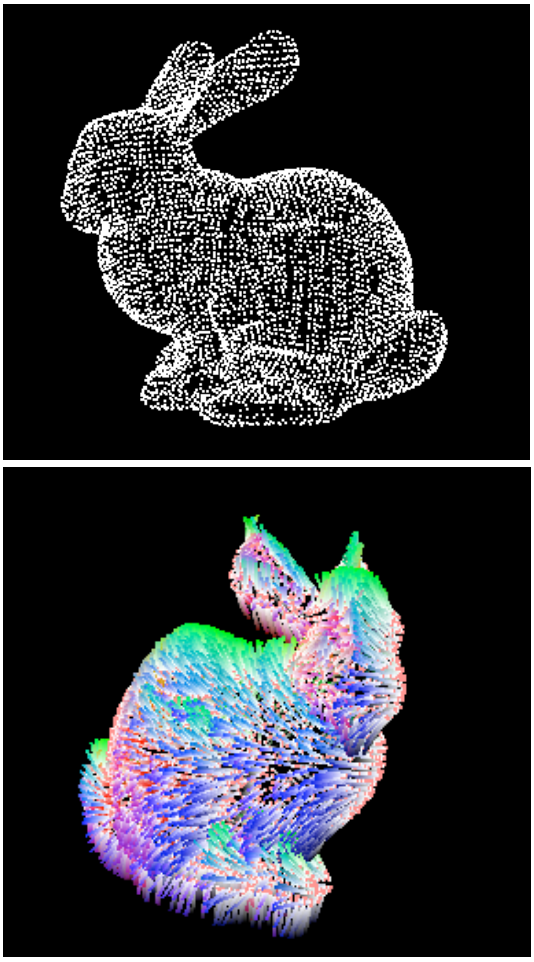

Figure 4. Top: Re-rendering the vertices using coordinates obtained from the vertex program. Bottom: Vertex coordinates and normals retrieved from a vertex program rendered using points and lines in order to show the direction of the normals

We can also render other data in the resulting image by setting the vertex color to the desired

variable, and rendering the resultant vertex buffer. To avoid the data being clamped to 0..1 we can assign the value to a different vertex attribute, such as any of the texture coordinates, and use a fragment program to return that parameter to the frame buffer.

5. Conclusion and further work

The method presented in this paper allows the user to return data from a fragment and vertex program with a small amount of modification to the original program. It is capable of analyzing both vertex and fragment programs, and returning meaningful data back to the user.

The area of greatest need of work is the seamless integration of this method with the original program. To make this method truly usable it must be possible to apply it without modifying the original application. This has been addressed in a separate paper, which incorporates both this method and other means of debugging the OpenGL pipeline.

Another area of future development is to incorporate more varied visualization models. This, however, is more dependent on the data being rendered, and may require minor coding to represent the data in a way which makes it meaningful for the specific application.

6. Bibliography

LINTON, M. A. 1990. The evolution of Dbx. 211–220.

SISTARE, S., ALLEN, D., BOWKER, R., JOURDENAIS, K., SIMONS, J., AND TITLE, R. 1992. Data visualization and performance analysis in the prism programming environment. In Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing, North-Holland, 37–52.

STALLMAN, R. M. 1989. GDB manual (the GNU source-level debugger). Tech. rep., Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617)876 -3296, USA, Jan. Third Edition, GDB version 3.1.

ZELLER, A., AND LÜTKEHAUS, D. 1996. DDD – a free graphical front end for unix debuggers. SIGPLAN Not. 31, 1, 22–27.