Dynamic Code Generation for Realtime Shaders

Niklas Folkegård*
Högskolan i Gävle

Daniel Wesslén†
Högskolan i Gävle

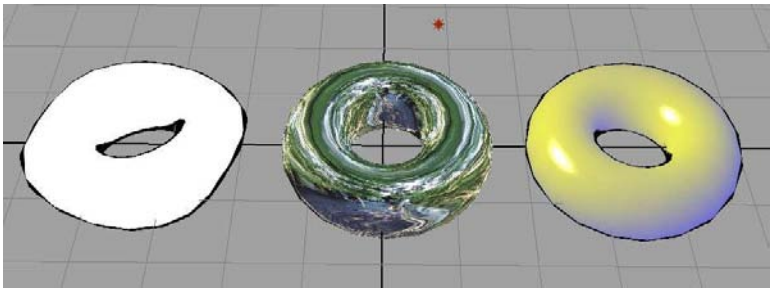

Figure 1: Tori shaded with output from the Shader Combiner system in Maya

Abstract

Programming real time graphics has been made more efficient by introducing high level shading languages such as Cg and GLSL. Writing shader programs can be redundant since the same functions appear in many shaders. This article suggests a method to combine single functions and create compound shaders in runtime. Redundancy is avoided by dividing programs into smaller, reusable parts. An algorithm is presented for joining these parts into working shaders based on just a few parameters.

CR Categories: I.3.0 [Computer Graphics]: General— [I.4.8]: IMAGE PROCESSING—Shading D.1.2 [Software]: Programming Techniques—Automatic Programming

Keywords: Graphics hardware, GLSL, GPU programming, programming efficiency, real time shaders, dynamic code generation, meta programming, computer graphics

1 Introduction

Real time graphics have become significantly faster and better with the introduction of programmable graphics cards (GPU:s) for the consumer market. The first of these GPU:s were released in 2001 [Fernando and Kilgard 2003]. Since these only could be programmed on an assembly level, the work of creating faster real time graphics became a lengthy and expensive procedure. In 2002, Nvidia Corporation released a high level shader language for programming this type of hardware, Cg (C for Graphics) [Mark et al. 2003], and during 2003 this was followed by the release of The OpenGL Shading Language, GLSL [Kessenich et al.].

*e-mail: nfd@hig.se
†e-mail: dwn@hig.se

With the introduction of high level shader languages, the work of creating high end real time graphics has become faster, easier and more comprehensible. But for large scale projects, where many different shaders are used, the process of writing shader programs still takes an unnecessary amount of time. The main reason for this is redundancy — even though many shaders share the same concepts, they still have to be written into each individual program. It would be preferrable if each concept only had to be written once, thus making the choice of concepts the relevant task in shader creation.

This article aims to present a solution where a complete shader program is created dynamically from short sections of code, representing the various concepts of shading algorithms. Hopefully, this solution will make using shader languages for implementation of hardware based real time graphics simpler and more efficient.

2 Background

2.1 Shader Languages

Most shader languages resemble C, but are specialized for simple computations involving vectors and matrices. The languages address the construction of the graphics hardware, where each vertex is sent through a pipeline where the position and material properties of the vertex are extracted and transformed to be used in calculating the colour of the fragments. Therefore the shader languages can affect the look of the graphics in two steps of the computations: before the vertex is transformed into fragments, and before the fragment is transformed into a pixel [Fernando and Kilgard 2003; Rost 2004].

Shader languages support run-time compilation. The application using shader languages as a support system for graphics computations can store the code as strings and call a built-in compiler when the program is needed. In most cases, just one program of each type may be on the GPU, but the application can easily swap shader programs in run-time [Fernando and Kilgard 2003; Rost 2004].

2.2 Automatic Creation of Shaders

One of the many benefits of shader languages like GLSL and Cg is that they make the borders between CPU and GPU based compu-

tations clear. In order to uphold this clarity some traditional methods for reducing code redundancy, such as procedural abstraction, must be omitted. Each shader program must be sent to the GPU as one unit, and as soon as it has started there is no good way to return to the CPU to call external functions. Because of these limitations, shader programs have typically been written as one big main-function where common concepts have been manually entered wherever they have been needed, thus bringing redundancy to any project of size.

Adding to the complexity of the problem, there is also the issue of combinatorics. Recent research has shown that it is possible to find faster computations for some cases of shading [Hast 2004]. This brings a need for a system able to handle many implementations of the same concept, and the ability to choose which implementation to use based on the properties of the rendering task at hand. Adding this kind of flexibility to a project with all hand-written shader programs would make it necessary to write duplicates of each shader using these concepts. The work load could easily become overwhelming when the number of alternative implementations increase.

There are ways to overcome the problem, however. By dividing the code of shader programs into smaller parts and providing some means by which these parts can be joined to form new programs, the work of creating shader effects can be made less redundant. Concepts can easily be defined in one place, and reused in many. While there are some implementations of this idea [Bleiweiss and Preetham 2003; LightWorks 20/05/2004; McCool et al. 2004; RTzen inc.], the solutions available do not fully encompass the following goals:

- code redundancy reduction
- ability to choose from many different implementations of the same concept
- ability to automatically include necessary, intermediate code
- clear borders between CPU and GPU

The solution presented in this article will show a way to overcome some of the limitations of established shader languages, without inventing a new language or restricting the artist to work with predefined algorithms.

3 Shader Combiner

This section will present a system that, based on a few parameters, automatically links parts of shader code together to form working shader programs. Manually created by the programmer, these shader code parts should be made globally available in the system. By specifying the criteria for a particular shader a working solution should be constructed by the system. If there are many solutions to the same query it should be possible to choose which one of the solutions to use. The implementation which this article is based on is made using GLSL, but the ideas expressed can easily be transferred to any shading language with similar properties.

3.1 Dividing Code into Parts

A shader program can be seen as consisting of operations that need to be performed in a certain order. Some operations set the value of certain variables, while others use these variables as part of their computations. Looking at shader programs this way, it becomes obvious that a program can be divided into parts as long as we can make sure that these parts appear in the right order whenever they are rejoined to form a new program. To ensure this it is necessary to provide each part with some properties that can be used for linking it to other parts. Taking into account that each part actually does something useful as well, we end up with three important properties that define a part:

1. inputs, or preconditions, representing the concepts needed by a part
2. computations, or operations, representing the actual work of a part
3. outputs, or postconditions, representing the result of the computations that might be useful for other parts

To add flexibility and ease of use, the implementation uses a larger set of properties for its parts, but these three properties are all that are conceptually needed. From this description it is possible to create parts representing the concepts needed for shading.

3.2 Combination Complexity

Today's programmable GPU's afford operation control in two stages of computations — in vertex computations and in fragment computations. Thus a solution could mean that two separate programs should be used, one in each stage. Consequently it is useful to consider a solution as being a pair of programs, where both parts of the pair should be used to reach the desired result. The system needs to make sure that two separate programs work without conflicting with each other, and it needs to do this even when none of the two programs is fully constructed.

A part can provide any number of outputs and require any number of inputs, a fact that could slow down the process of construction. In a small project there might be just one part for each concept, making the work of combining the parts easy. For larger projects, however, there could be many different parts that all need to be considered by the system. The many parts and their many connections can also form circular dependencies, where part *A* needs input from part *B*, while *B* needs input from *A*, thus making an infinite loop.

3.3 Rejoining Code Parts

To reach a solution the user should state the conditions to be met by the program pair. The system will then search through the available parts and return a program pair fulfilling the conditions stated. In order to achieve this we need an algorithm $getShader(C)$ that from a set of conditions C will return nought or one working program pair. This section will present such an algorithm.

3.3.1 Solution parts

The operations in each part are specific for a certain state in the programmable graphics pipeline. A fragment part can only fulfill conditions for other fragment parts, while a vertex part can fulfill conditions for other vertex parts as well as for fragment parts. In the future other parts of the pipeline are likely to be accessible for programming and parts of a future type will thus have their own rules for how they affect other types in the system. However, to simplify this explanation we just note that there is a difference between how different types of operations affect each other, and assume that the system can handle these rules in a way invisible to the user.

The user can provide the system with solution parts as he/she finds necessary, and these parts will be used by the system. Note that the conditions are global, e.g. if a part P_1 has a precondition called *normal* and a part P_2 has a precondition that is also called *normal* they will both benefit from a part P_3 having a postcondition called *normal*. Therefore the parts can be connected to each other, so that one or more parts fulfill all preconditions of another part. A part having no preconditions is complete and will need no more help from the other parts in the system. Listing 1 shows a selection of solution parts from the implementation.

Listing 1: Solution parts

```
#vertex 10 "stdVertex" manual
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
        out vec3 viewer=normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
        out vec3 vNormal = normalize(gl_NormalMatrix * gl_Normal);

#vertex 10 "stdDiffuse" manual
        out vec3 diffuseMat =  gl_FrontMaterial.diffuse.rgb;

#vertex 10 "stdLight0" manual
        out vec3 light0Pos = gl_LightSource[0].position.xyz;
        out vec3 diffuseLight = gl_LightSource[0].diffuse.rgb;

#vertex 10
        in vec3 viewer;
        in vec3 light0Pos;
        out vec3 vLight0 = normalize(light0Pos - viewer);

#vertex 10
        in vec3 vLight0;
        in vec3 vNormal;
        out float vNdotL = dot(vLight0 , vNormal);

#vertex 10 "stdLambert" manual
        in float vNdotL;
        in vec3 diffuseMat;
        in vec3 diffuseLight;
        out vec3 lambert = vec3(diffuseMat * max(0.0, vNdotL) *
                diffuseLight);

#fragment 100 "fLambertOut"
        in vec3 lambert;
        gl_FragColor = vec4(lambert, 1.0);
```

3.3.2 Representation of the problem

Knowing the structure of the solution parts, it is now possible to represent the shader program requested by the user. If we take a solution part P and let its preconditions consist of the conditions C specified by the user (the caller of the algorithm $getShader(C)$), and in all other aspects leave this part empty, we will have a solution part demanding the functions specified by the user but not doing anything in itself. If we perform a search through the system using this part as a starting point, it will be connected with other parts fulfilling the preconditions. When the part is complete we have a solution.

In reality it is not all that simple. As mentioned earlier today's hardware have two programmable stages — the vertex stage and the fragment stage — and since we do not know which type of solution part will fulfill certain preconditions there is always a risk that the system returns two separate programs. Moreover, these programs must work together in an optimal way (for example, as many operations as possible should be performed in the vertex stage since this stage is used less frequently). Since the desired solution may be a pair of vertex and fragment programs, the problem will throughout this explanation be called a *program pair*. We also note that, like the solution parts, a program pair having no preconditions is complete.

3.3.3 Linking parts into a solution

Initially, a program pair is created whose preconditions correspond to the user's demands. The system can now search for solution parts that fulfill these preconditions. If it is possible to connect this program pair with solution parts fulfilling all initial preconditions we have a *possible* solution, but new preconditions may arise from the parts found and it is not until all preconditions are fulfilled that we have a complete solution.

The parts that are found helpful are connected to the program pair by letting the preconditions of each helpful part P become part of the preconditions of the program pair. At the same time, the postconditions of each part P eliminate preconditions of the program pair. By this operation we get a new set of preconditions affecting how future search is done. If the set of preconditions is empty, we consider the program pair complete.

3.3.4 Finding possible solutions

Finding suitable solution parts should be fast and easy. Each time a solution part is found, new preconditions might be added, and each time multiple parts fulfill the same precondition the search path would branch. The best thing would be if we quickly could find a set of solution parts fulfilling all preconditions that are currently unfulfilled. This could be done by mapping each available postcondition in the system to the corresponding parts.

If the system only has one solution part for each entry in such a map, a list of preconditions could easily be converted to a list of matching solution parts. But the system could have mapped any number of parts in each entry. We need to find the different combinations of solution parts fulfilling all current preconditions. The internal order of the elements is of no importance. In this stage we are only concerned with whether a part is *useful*, not *in which order* it will be used.

If we do not find a set of solution parts fulfilling all preconditions of the program pair, we will not find a solution. If we find many sets, the search will branch.

3.3.5 Putting operations together into working programs

The program pair is built as a collection of solution parts needed to make it complete. As soon as it is, we need to transform this collection into working shader code. Each part has an element representing its operations, and this is now used to build the final shader programs. From the collection, all parts having no preconditions are added. The added parts provide some postconditions and gives us a new state. Now we can add all collected parts whose preconditions are fulfilled by this new state. This process goes on until the collection is emptied.

Note that this method will not be able to put together parts with circular dependencies. This behavior is desired, since such dependencies mean that the code is inoperable. If we in any step cannot put in new code we have a case where the parts cannot be joined into working code, and the current solution will fail. In this stage it is also possible to perform other tests on the code (e.g. we might want to try and compile the code, or test it against the state of the calling application).

If the search has branched, we will end up with many program pairs. What remains is selecting one of these solutions. Here it is possible for the user to decide the criteria to be used in the selection process. If no criteria are provided, the system will pick whichever solution it regards as the fastest, based on the sum of part costs.

3.3.6 The final algorithm

The algorithm initially sought for, $getShader(C)$, is now easy to define (see Algorithm 1). It will search the system, generate as many program pairs as possible and choose one of them to return.

Algorithm 1 $getShader(C)$

procedure GETSHADER(C)
  Construct a program pair α_0 whose preconditions are C
  for each α_n in the system **do**
    while α_n has preconditions **do**
      Find all sets of solution parts whose postconditions fulfill all preconditions of α_n
      for each set found **do**
        Add a new program pair α_m to the system
        Eliminate all preconditions of α_m
        Add the parts in the set to α_m
        Add the preconditions of all parts in the set to α_m
      end for
    end while
  end for
  for each α_n in the system **do**
    while α_n has parts **do**
      Mark all added parts whose preconditions are fulfilled by the postconditions of α_n
      Add the postconditions of all marked parts to α_n
      Add the code of all marked parts to α_n
      Eliminate all marked parts from α_n
    end while
  end for
  return the best α_n generated
end procedure

3.3.7 Public and explicit parts

In the description above, a global set of solution parts is assumed for the search. Cases may arise where some solution parts are not useful, even though they provide functionality demanded by the user. For instance, all variables built into GLSL assume a standardized format, and each solution part providing functionality by using these variables belong to a set of solution parts not suitable for global accessibility since we cannot assume that all objects rendered are standardized.

The solution to this problem is simple. In the previous we have assumed that searching is done in a global set. Now this set is divided into a *public* set consisting of parts that can always be used, and a *private* set consisting of solution parts that can only be used when they are explicitly asked for. Searching will thus be done in the public set and the set of parts explicitly asked for.

It is also possible that a user needs certain parts to be included in a solution. This too is easily attended to. We simply connect these parts to the initial program pair before starting the search. That way the preconditions will be updated to represent a stage where the demanded parts are included.

3.4 Implementation

3.4.1 Description

The implementation [Folkegard 03/06/2004] is GLSL specific, which most of all is seen in how it handles types and modifiers of global variables. In the algorithm, handling global variables has been avoided, but in the implementation it is an important part. Values assigned through *const* or *uniform* do not require operations in the shader program, and are therefore regarded as the fastest method. Vertex shader values used in a fragment shader must be passed by declaring *varying* variables. The implementation takes care of this handling.

The user writes the code for the parts in a simple syntax where preconditions, global variables, operations and postconditions are clearly indicated. The parts are then added to the system and stored in structures supporting fast finding. Searching is done by specifying three parameters (a,b,c). The parameters are lists of requested functions (a), explicitly demanded parts (b), and parts that should be made available in searching (c). The system can return strings with the code found by the system, as well as lists of global variables. Output from the implementation can be found in listings 2 and 3.

Listing 2: Simple Lambert Vertex Shader Automatically Created

```
varying vec3 lambert ;
void main()
{
        vec3 diffuseMat;
                diffuseMat = gl_FrontMaterial.diffuse.rgb ;
        vec3 vNormal;
        vec3 viewer;
                gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
                viewer=normalize(vec3(gl_ModelViewMatrix * gl_Vertex));
                vNormal = normalize(gl_NormalMatrix * gl_Normal) ;
        vec3 diffuseLight;
        vec3 light0Pos;
                light0Pos = gl_LightSource[0].position.xyz ;
                diffuseLight = gl_LightSource[0].diffuse.rgb ;
        vec3 vLight0;
                vLight0 = normalize(light0Pos - viewer) ;
        float vNdotL;
                vNdotL = dot(vLight0 , vNormal) ;

        lambert = vec3(diffuseMat * max(0.0 , vNdotL) * diffuseLight) ;
}
```

Listing 3: Simple Lambert Fragment Shader Automatically Created

```
varying vec3 lambert ;
void main()
{
        gl_FragColor = vec4(lambert, 1.0);
}
```

3.4.2 Search performance

The implementation has been run with 1 to 5 different variants for 8 parts (totalling 8 to 8^5 solution parts). The tests have been made on a computer with 2.00 GHz Intel P4-processor and 1 GB RAM, and show that the system can get a solution within $\frac{1}{50}$ of a second if all parts have two different variants. With three variants the search will give a small but noticeable decrease in rendering speed (see table 1). It should be noted that it is not very likely that many parts have different variants. Nor is it likely that searching will be done inside a critical render loop.

Parts	Variants of each part	Run time in seconds
8	1	0.156
8	2	2.109
8	3	11.907
8	4	43.719
8	5	128.719

Table 1: Search time for 100 searches with varying number of part variants

3.4.3 Maya extension

The implementation has also been added as an extension to Alias Systems Maya 6.0 [Folkegard 30/09/2004], allowing a user to work directly with hardware shading in the Maya work environment. The extension connects the Shader Combiner system to the Maya user interface. Solution parts can be created, viewed and selected to generate the desired effects on polygon objects. The result of each generated shader is immediately visible in the work environment, and the final code can be read and exported from within Maya. All global variables of each shader can be connected to the rest of the Maya system, including the expression and scripting engine and the animation functions. The main use for this extension is as a tool for graphics algorithm artists developing real-time effects for games and research purposes, but it can also be used to enhance hardware rendered effects for film and TV.

4 Discussion

The algorithm is sufficient for the purpose of finding working program pairs for real time shaders. By focusing only on solution parts useful at a given moment the algorithm works faster than it would have done if it had gone through all existing parts. The algorithm can improve by rejecting unsuitable solutions before all solutions are found, but finding such an optimized search algorithm is outside the domain of this work.

The system is well suited to quickly create shaders. For easy control of the result it has been useful to create a node representing the preconditions and final operations of the result, and then to require that part in all solutions. Since the system generates directed acyclic graphs, the user is urged to design shaders in the same fashion. That way the design is easier to grasp, and circular dependencies are avoided.

4.1 Future development

For substantial usefulness in large scale development of real time shaders, the system needs to be independent of language. The search algorithm should be modified so it rejects unsuitable solutions in earlier steps of the search, thereby avoiding the problem of huge time costs when encountering many alternative search paths. Shaders being too large to run in hardware could also be divided by the system, much as the method suggested by Chan et. al [Chan et al. 2002].

Currently all solution parts consist of static functions. If output from one shader shall be used as input for another, the receiving shader must state this. In order to be fully flexible the system should be able to generate compound shader trees where the connection between nodes and their internal order can be specified arbitrarily. This can be made possible if the solution parts are allowed to receive and give out arbitrary parameters, which would mean that the system creates its own static functions from a description of a general function. If this succeeds there is a possibility to use the system to assemble other data, for example regular procedural code in systems for visual programming.

5 Conclusion

Shader programming can be made less redundant by dividing programs into parts. Thereby hardware shading is made more modular and more flexible. By using the shader combiner system presented here, an application can use a richer set of real time shaders without having to put a lot of time into developing these. Dividing shader programs in smaller parts can make the different concepts of shading very clear. Work can thereby be focused on using the concepts for one's own artistic purposes and for quickly trying out new shading ideas. The implementation has a sufficient speed for efficient use in real-time applications, making it useful for both shader development purposes and as a subsystem in games and visualizations.

References

BLEIWEISS, A., AND PREETHAM, A., 2003. Ashli – Advanced Shading Language Interface. http://www.ati.com/developer/eurographics2003/AshliViewerNotes.pdf.

CHAN, E., NG, R., SEN, PRADEEP, S., PROUDFOOT, K., AND HANRAHAN, P. 2002. Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware. In *Graphics Hardware(2002)*, 69 – 78.

FERNANDO, R., AND KILGARD, M. J. 2003. *The Cg Tutorial*. Addison–Wesley.

FOLKEGARD, N., 03/06/2004. Shadercombiner implementation. http://sourceforge.net/projects/shadercombiner/.

FOLKEGARD, N., 30/09/2004. Hardware Shader Combiner for Maya. available through Creative Media Lab, Gävle.

HAST, A. 2004. *Improved Algorithms for Fast Shading and Lighting*. PhD thesis, Uppsala Universitet.

KESSENICH, J., BALDWIN, D., AND ROST, R. The OpenGL Shading Language.

LIGHTWORKS, 20/05/2004. Programmable Hardware Shading In Applications – Challenges and Solutions. http://www.lightworkdesign.com/news/media/ProgrammableShadingInApplications.pdf.

MARK, W. R., GLANVILLE, S. R., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. In *ACM Transactions on Graphics*, ACM, 896 – 907.

MCCOOL, M. D., DU TOIT, S., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader Algebra. In *ACM Transactions on Graphics*, ACM, 787 – 795.

ROST, R. J. 2004. *OpenGL Shading Language*. Addison–Wesley.

RTZEN INC. RT/shader.