# An Effective Composition Method for Novice Shader Programmers

**Q. Hendrickx**
Computer Graphics Group, Delft University of Technology, The Netherlands

**R. Smelik**
Modelling, Simulation & Gaming Department, TNO, The Netherlands

**R. Bidarra**
Computer Graphics Group, Delft University of Technology, The Netherlands

## ABSTRACT
Shader programming has become an increasingly important and complex task in computer graphics. Because of the in depth knowledge required to write effective and efficient shader programs, novice users are finding it increasingly harder to construct these programs manually. As a result, several tools have been developed to assist developers in producing shader programs more efficiently and easily than writing them with traditional text-based editors. Graph-based composition tools have been proposed as an attractive alternative, as shader programs are intuitively represented as a graph structure. However, such tools do not usually exploit the structural and semantic information implicitly available in the graph to assist building, understanding and debugging shader programs.

This paper presents a composition method that allows novice shader programmers to quickly and interactively prototype shader programs. This method takes advantage of a graph based composition, allowing developers to design their shaders at various levels of abstraction, effectively promoting node reusability. Furthermore, consistently building upon the graph's structure and semantics, the method significantly eases shader program debugging and provides very helpful insight into its workings, as demonstrated by the results of the open source prototype system implemented.

It is concluded that this method provides an efficient and very intuitive alternative for developing shaders. It is therefore particularly appropriate for novice shader programmers and students.

### Author Keywords
Graphics shaders, Novice users, ShaderComposer

### ACM Classification Keywords
I.3.4 Computer Graphics: Graphics Utilities

## INTRODUCTION
Starting with the introduction of programmable shaders in 2001, developers of graphics applications are given more and more freedom to customize the rendering pipeline. However, developing shaders is often complex and cumbersome. Debugging shader code is hindered by the fact that, without specialized tools, it is not possible to introduce breakpoints, step through the code, or inspect values of local variables. Therefore, the correctness of a shader program typically has to be assessed by inspecting its visual output. Furthermore, the parallel execution of shaders makes it difficult to predict the performance impacts of specific changes.

Most of the time, professional shader programmers are experienced enough to overcome most of the aforementioned issues. However, when someone is just starting to learn writing shader programs, these problems can become major obstacles. Often, students will not yet be familiar with the parallel execution and the GPU rendering pipeline, hindering them to learn the principles behind specific shading effects.

To solve this problem, several tools have been introduced that aid developers in designing shaders. For example, AMD's RenderMonkey and nvidia's FX Composer provide real-time previews of the shader's output, while editing the code [2] [7]. However, these tools are still based on a text-based editor, requiring the developer to directly write shader code. While current high level shader languages are somewhat based on the C programming language, there are a lot of important differences that beginners will have to learn before they are able to write correct and efficient shader code.

A first attempt to increase the efficiency of writing shader programs was made by [4]. It introduced a system in which shader programs are represented in a structure called a *shade tree*. Every node in this shade tree represents an operation, the edges between the nodes are parameters. Every node produces one or more output parameters and can use zero or more input parameters. This model has proven to be very adequate for describing shader programs, as even today most shader programs can still fit into this model. However, [4] does not construct the shader program as a tree, instead the tree is generated based on a written shader program. This is different in [1], where a graphical implementation of a shade tree is introduced. This allowed users to construct shader programs through a simple user interface without requiring them to learn a specific shading language.

A somewhat different, but also very intuitive, visual representation of a shader is a directed acyclic graph. Inputs, mathematical operations, texture sampling and outputs are represented as nodes and the edges represent the flow of control. The editor of the popular game engine Unreal provides such a graph-based editor for defining material shading [3]. This allows designers to develop shaders in an accessible way, without writing a single line of code. However, this editor does not take full advantage of the graph's structure and semantics, e.g. to aid the designer in debugging or optimizing the shader.

In [6] a similar system is introduced but its focus is directed towards high-level features such as shadows and reflection. By inspecting the graph, connections between nodes are automatically inferred, ensuring that the generated code will be free of type-mismatches. However, no attempt is made to assist the inexperienced user in debugging the shader or providing information about intermediate result values.

A shader development tool specifically designed for use by non-programmers is introduced in [5]. The intended target users are artists and students. In a test study the response toward this tool was positive, with users indicating that it allows them to quickly create different shaders and receive good visual feedback.

The composition method presented here, and its prototype implementation, improve on the existing methods by utilizing the graph's structure and semantics for two novel goals: (i) it provides great insight into the inner workings of graphics shaders and, is therefore a valuable learning tool, and (ii) it improves the designer's efficiency, significantly shortening the iteration cycles of shader development.

## SHADER COMPOSITION METHOD

A graph structure is a natural representation for visual shader programming. In this section we will show how this structure not only provides an intuitive visualisation of shader flow, but gives many opportunities for exploiting its structure and semantics to help designers writing shader programs.

In our method, shader programs are represented as a directed acyclic graph. Each node in this graph can have multiple input and output variables, which provide the connection points for the graph's edges. The type of node defines an operation that computes the output variables, based on its input variables. Example operations include a dot product, value clamping or a texture sample fetch. By connecting outputs of nodes to inputs of other nodes we can compose graphs that define complex operations. A node with no inputs can represent either a constant value or a per vertex varying variable (passed on from the vertex shader). The result of the shader program is represented by the output node; typically this value defines the fragment's colour.

A common pattern in programming is *abstraction*, i.e. providing a simple interface to complex functionality. This pattern is also very useful in shader development. For example, although one might want to use the phong lighting model in a shader, often one will not be interested in its specific implementation. In our method, we allow for this kind of abstraction using the notion of abstract nodes. A sub-graph that contains nodes whose input and output variables have not all been connected can be collapsed into a new type of node that exposes all those input and output variables. This idea is illustrated in Figure 1, where a sub-graph containing several operations is collapsed to form an abstract *Phong Lighting* node. A designer needs only to supply the proper inputs for the *Phong Lighting* node to obtain its output, without any concern for of its implementation. It is possible to go further with this abstraction, effectively constructing a hierarchical node tree, where high-level nodes are composed of several connected abstract nodes.

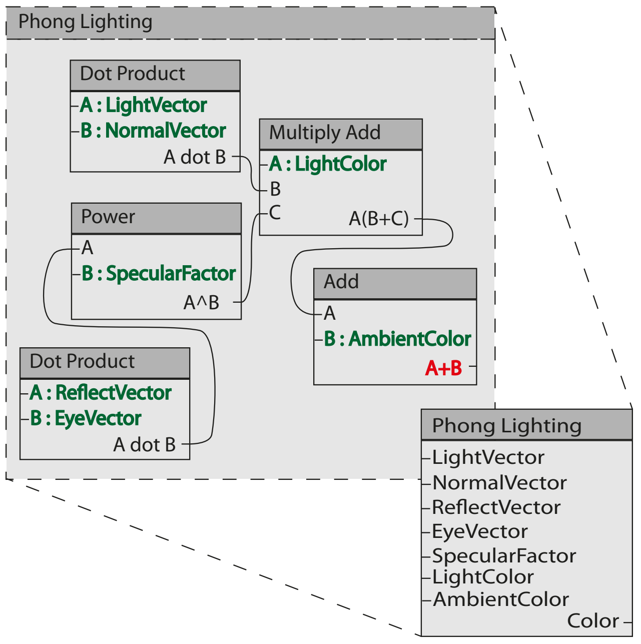

**Figure 1. A phong shading fragment designed as a graph. A collapsed node based on this sub-graph would expose the connections indicated in green as input parameters and the connections indicated in red as output parameters.**

For shader programming, it would be helpful to be able to inspect values of local variables and intermediate results. However, in a text-based environment, designers only have access to the shader's final output. We provide the designer with a real-time view of the intermediate result ouput by any graph node. Continuing the phong lighting example, Figure 2 shows relevant intermediate outputs of the phong shader, composed as shown in Figure 1. In debugging, a designer could use this feature to quickly identify which part of a shader is not working correctly.

It is straightforward to implement this inspection functionality, using the shader graph structure. For this, we clone the original graph several times, and for each of these graphs we directly link a different intermediate node's output to the out-

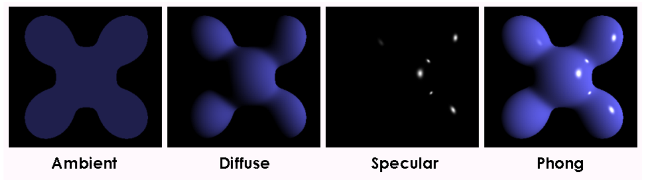

**Figure 2. Different intermediate outputs of a phong shader that can be used to debug and gain new insights in the working of a shader.**

put node. Then, we compile the resulting shaders and render the scene for each of the different shaders. The resulting images provide helpful insights into the workings of the shader. Sampling specific pixels from these images allows designers to debug the shader in even more detail.

Often one would like to compare different variations of the same shader, such as slight changes in some constant or a different approach at calculating a specific value. By keeping track of the shader version history, we allow the designer to quickly compare several versions of the same shader. For this, we store a screenshot and a performance analysis for each version of the shader, thus allowing designers to make trade-offs between render quality and performance. In a graph structure, edit actions occur at a high level, making it easy to identify only the valid versions of the shader.

## SHADER COMPOSER IMPLEMENTATION

The shader composition method described above was implemented in a stand alone design tool, Figure 3 provides an overview of its GUI. In the top-left area, the shader graph can be composed. Using the available tabs it is possible to inspect the shader in different representations, such as XML or as compiled shader code in one of the dominant languages. At the right hand side there is an interactive preview that updates after any changes made to the shader graph. At the bottom we visualize the version trail tree; selecting a previous version reverts the shader to this previous state.

It is possible to debug the shader for a specific pixel by clicking in the shader preview window. The output value of this pixel is shown underneath the preview window in a text based format. Additionally, it is possible to check the value of this pixel at each connection between nodes in the graph. In effect, this provides an instant debugging experience that is comparable to step by step debugging that is available when using regular programming languages. This feature allows users to quickly trace through the computations and identify any problems.

As opposed to inspecting a specific pixel of the shader output it is also possible to view all pixel values at a specific location in the graph. By hovering over a connection between two nodes, a popup window opens that shows the output of that particular node. This popup window can be pinned so that it remains visible even when the cursor is no longer hovering over the connection. Because the output type of a node isn't always in the visual range, we sometimes have to convert it before showing the output. For example, a single channel float value is converted into a grayscale representation and boolean variables are converted into black and white values. These preview windows are very helpfull when trying to grasp the meaning of intermediate shader variables (e.g. specular factors or view space normals).

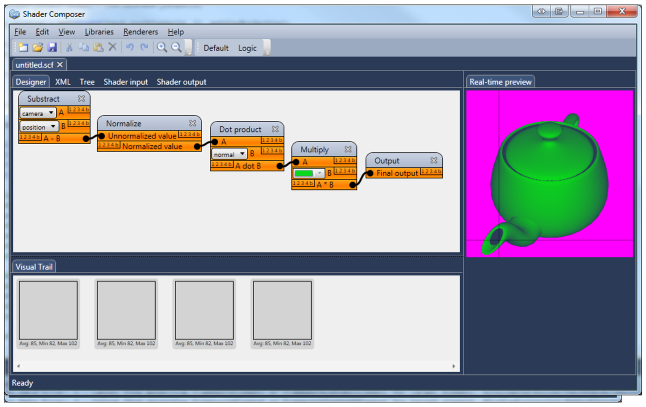

**Figure 3. GUI of the developed shader tool. A live preview is visible in the top right corner, the bottom panel is used to visually track changes made to the shader.**

Whenever changes are made to the shader graph it is possible to create a new tag. This will create a new entry in the version trail at the bottom of the window. Every entry in the version trail contains a screenshot of the final shader output and provides statistics about the performance of the shader. This allows users to compare the quality of different shader configurations as well as the performance. Especially for novice users this is an important feature because it is not always obvious what the perfomance impact of specific operations on the GPU might be. Sometimes approximations that sacrifice quality over performance can be the correct choice in graphics programming.

By clicking on an entry in the version trail it is possible to revert to a previous version and start editing the shader in a new branch. This approach uses the idea of version control systems such as SVN, and extends it with easy comparison between shaders based on quality and perfomance.

The shader development tool, which implements the discussed composition method, is open source and can be downloaded at `http://code.google.com/p/shadercomposer/`.

## RESULTS

We discuss the results of our composition method using a real life scenario of a terrain shader. Figure 4 shows a screenshot of a terrain rendered using the shader we are about to discuss. The corresponding shader graph is shown to the left.

Before we start designing the shader graph, we first have to select the terrain characteristics that influence the terrain soil type. Here, we use only elevation and slope to keep this example easy to understand, but this can be extended to a wide variety of parameters, e.g. the local climate, distance to a water source, occlusion from the sun or the distance to urban areas. Next, we determine the terrain soil types.

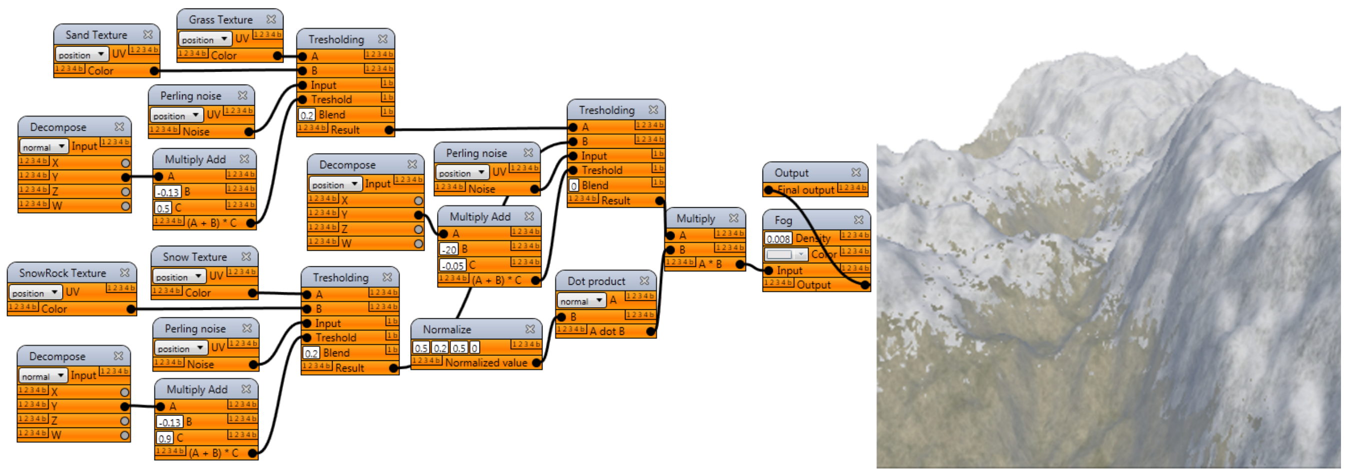

**Figure 4. Screenshot of terrain rendered using a shader developed using the proposed composition method. A global overview of the corresponding shader graph is also shown.**

The elevation parameter separates the terrain in two parts: mountaintops are to be covered in snow, while the valleys are filled with grass. At steep angles snow will not accumulate as much as on flat terrain, therefore the slope parameter will further influence snow coverage, exposing rocky terrain on steep slopes. Each soil type is modelled in a separate sub-graph and stored as an abstract node. By combining multiple detail textures, we can achieve more variation within a single soil type.

After this, we require a new node to blend these types together at transition boundaries. A simple linear blend does not produce convincing results, therefore we create a threshold blending node. The soil type that has the highest weight will be fully visible, while the other terrain type will not be visible at all. Before being compared, the weights of both soil types are slightly perturbed by a Perlin noise texture. This produces discrete but organic transitions. The idea of a threshold blending is encapsulated in an abstract node, that can be saved for reuse in other projects.

## CONCLUSION

In order to make shader technology accessible to a much wider public, better tools are needed which effectively foster shader understanding and facilitate shader debugging. With this motivation, we developed the presented graph-based composition method. The method is innovative in that it takes advantage of the graph hierarchical composition, allowing developers to design their shaders at various levels of abstraction, effectively promoting node reusability: sub-graphs, or whole shaders, that have been completed and tested, can be encapsulated in a single node, stored, shared and freely reused at a later stadium.

Another advantage of this method is that it makes use of the shader generator to expose the very nature of the shader graph's structure and semantics, making it very easy and intuitive to probe the output of any node in the graph and *see* the corresponding results. This facility provides very helpful insight into shader program workings, and significantly eases its debugging.

We developed an open source prototype system that implements this method and demonstrates all its interactive features described here. A shader code compiler takes care of exporting the shader program to the shader language of choice (e.g. HLSL, CG and GLSL), so that it can be used in some external application.

From our experiments and test sessions, we can conclude that this method provides an efficient and very intuitive alternative for developing shaders. It is therefore particularly appropriate for novice shader programmers, interested students, or within the context of introductory graphics courses.

## REFERENCES

1. Abram, G. D., and Whitted, T. Building block shaders. *SIGGRAPH Comput. Graph. 24* (September 1990), 283–288.

2. AMD. RenderMonkey, 2008.

3. Burke, D. *A tutorial on creating materials in the Unreal Editor*. Epic Games, 2009.

4. Cook, R. L. Shade trees. In *Proceedings of the 11$^{th}$ Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, ACM (New York, NY, USA, 1984), 223–231.

5. Fitger, M. Visual shader programming. Master's thesis, KTH Royal Institute of Technology, 2008.

6. McGuire, M., Stathis, G., Pfister, H., and Krishnamurthi, S. Abstract shade trees (preprint). In *Symposium on Interactive 3D Graphics and Games* (March 2006).

7. Nvidia. FX Composer 2.5, 2008.