Automated Combination of Real-Time Shader Programs

Matthias Trapp[†] and Jürgen Döllner[‡]

Hasso Plattner Institut, University of Potsdam, Germany

Abstract

This work proposes an approach for automatic and generic runtime-combination of high-level shader programs. Many of recently introduced real-time rendering techniques rely on such programs. The fact that only a single program can be active concurrently becomes a main conceptual problem when embedding these techniques into middleware systems or 3D applications. Their implementations frequently demand for a combined use of individual shader functionality and, therefore, need to combine existing shader programs. Such a task is often time-consuming, error-prone, requires a skilled software engineer, and needs to be repeated for each further extension. Our extensible approach solves these problems efficiently: It structures a shader program into code fragments, each typed with a predefined semantics. Based on an explicit order of those semantics, the code fragments of different programs can be combined at runtime. This technique facilitates the reuse of shader code as well as the development of extensible rendering frameworks for future hardware generations. We integrated our approach into an object-oriented high-level rendering system.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors; I.3.6 [Computer Graphics]: Languages; I.3.8 [Computer Graphics]: Applications; D.1.2 [Programming Techniques]: Automatic Programming

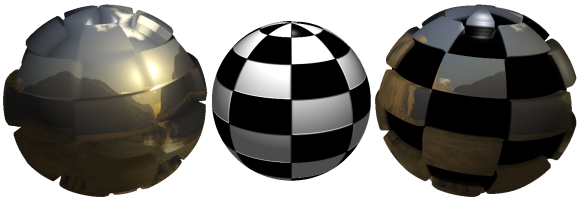

Figure 1: *Example for the automated combination of displacement and normal mapping shaders.*

1 Introduction

Today, programmable graphics hardware comes along with a main conceptual problem: it is only possible to have a single active shader program that replaces parts of the fixed function rendering pipeline and becomes part of the rendering context [Mar06]. This usually results in a large number of independent shaders for multiple variations of rendering. Based on current technology, individual shaders cannot be combined automatically because there are neither explicit shading language features for shader combination nor do typical shaders allow us to easily identify and reuse their functionality. This often requires a skilled programmer who is able to develop a new shader, whose functionality is a combination of the respective features. This aspect is an antagonism to the generic characteristics of a scene-graph based high-level graphics API.

The effective use of shader programs in a high-level graphic middleware includes the ability to reuse them and requires the combination or nesting of their functionality. Each program covers three areas of functionality with each being represented by a single specialized shader: vertex, geometry, and fragment shader. Reusing such programs saves expenses for repeated optimizations and debugging. This cannot be accomplished with existing data structures that bridges the gap between designer and programmer [MSPK06]. If we want to use shaders in a stand-alone way as well as to derive combined variants (Fig. 1) within a scene graph based rendering system, one encounters the following two problems:

- **Permutation**: In current engines or frameworks many shaders are variations and combinations of basic function-

[†] matthias.trapp@hpi.uni-potsdam.de
[‡] doellner@hpi.uni-potsdam.de

ality (e.g., material LOD approximations [OKS03], lighting models, animation, skinning). Creating and managing these permutations efficiently are two key requirements for systematic and cost-efficient development of real-time high-end 3D applications.
- **Independence**: Modern shader-driven engines [Ben02], frameworks or middleware utilize the concept of a shader library [AV02]. To achieve a generic solution, one has to ensure that multiple instances of shader permutations can interact and perform independent from each other.

The permutation problem can be solved by generating various shader source codes from source code fragments [McG05]. These fragments cannot be executed independently. The approach is limited to a known number of shader fragments and combinations. The so-called uber-shader [Har05] or super-shader is designed to reduce costly context switches of driver and hardware, which occurs when a shader program is loaded or replaced.
Our basic idea is a generic approach for uber-shader construction. It can combine shader programs that are composed of several shader source code fragments ("shader handler"), each with a predefined semantics. These programs can be executed independently and combined by invoking their handlers according to a given order of semantics. To enable such dynamic combination, our approach consists of two separate processing steps:

1. A preprocessing step for each shader program analyzes a tagged source code and transforms it into an intermediate representation.
2. In the second step, these intermediate representations are combined into a new shader program. For this, all intermediate representations are concatenated, and an additional shader is generated that controls the execution of the particular code paths.

Our main contribution is the dynamic accomplishment of the second step at runtime and comprises an analysis of control parameters for combined programs. In addition, we apply our approach to the high-level shading language GLSL [Kes06].
This paper is structured as follows: Section 2 shortly discusses related work. Section 3 explains the basic concept of shader handlers. Section 4 describes their combination and Section 5 exposes the execution mechanism. Section 6 concludes the paper and outlines future work.

2 Related Work

Hargreaves [Har05] describes how to automatically generate large numbers of shader permutations from a smaller set of input fragments. McGuire [McG05] demonstrates a shader framework that is able to render several effects. It permits arbitrary combinations of rendering effects to be applied to surfaces simultaneously. It uses runtime code generation to produce optimized shaders for each surface and a cache to reuse shaders for similar surfaces. [MSPK06] presented a system for authoring complex GPU programs through automatic combination of primitive shading functions. In [FW04] a method is suggested that combines single functions and creates compound shaders in runtime. In [BFH*04] a compiler and runtime system is presented that enables the combination of stream kernels. In [MDTP*04] this system is used to implement connection and combination operators for shader programs.

3 Handler Concept

To enable the generic combination of shader programs, it is necessary to provide additional information [MSPK06] for each shader handler. This information can be evaluated at runtime and is used to generate a combined shader program as well as to specify the behavior of the individual handlers. The most important information is the semantics of the handler in a given context. The semantics denotes the functionality of a shader. In addition, information regarding the runtime behavior of shader handlers with the same semantics must be provided.
The assignment of such information can be implemented by tagging the source code or by introducing new keywords to a particular shading language. We select GLSL as testbed for our approach. To assign a handler semantics to a source code fragment we choose to extend the target language with an additional keyword. Later, a preprocessor can eliminate this syntactical overhead. The following concept can be applied to all shader types (fragment, vertex, and geometry shader).

3.1 Prototypes

Usually, shader performs successive tasks, each of which are well distinguishably [FW04] (e.g., lighting, transformation, texturing, clipping). We denote such a task as *prototype* (*P*) and assume that a shader can be decomposed into a sequence of code fragments that are instances of them. A sequence of prototypes is denoted as *prototype list* (*PL*).

3.2 Shader Handler

A *shader handler* (*SH*) is a source code fragment of a specific prototype. Accordingly, we redefine a *shader* as a set of shader handlers. On this basis, multiple shader programs can be combined and nested by merging their particular shader handlers. Furthermore, a handler possesses an *execution mode* that defines its invocation at runtime. We identified four different execution modes for a scene graph based system:

- **Global:** The handler will always be active during execution of the uber-shader program.
- **Local:** The handler is only active if the corresponding shader program is active.
- **Optional:** The handler will be active if no handler of the respective prototype is already active.
- **Explicit:** The handler will be invoked if the corresponding shader program is active. Other handlers of the same prototype will not be invoked.

The following example contains three GLSL vertex handlers that represent a elementary functionality. In this case

Figure 2: *Compositing of different effects rendered by nested shader programs. Phong-shading is combined with texture mapping, normal mapping, and an x-ray shading effect (from left to right).*

the global handlers `onInit` and `onFinish` apply the language specific state (`gl_Vertex`) to the context and vice versa. They can be placed in the root node of a scene graph. The `onTransform` handler applies the viewing and projection transformation to the input vertex and will only be invoked if no other transformation handler is active:

```
handler global onInit(interface i) {
  i.Position = gl_Vertex;
}
handler optional onTransform(interface i) {
  i.Position =
    gl_ModelViewProjectionMatrix * i.Position;
}
handler global onFinish(interface i) {
  gl_Position = i.Position;
}
```

3.3 Handler Interface

To enable the nesting of shader handlers, as demonstrated in Fig. 2, it is necessary that shader handler can interact in a certain way. This process is formulated as reading from and writing to a global context. The structure of this context should contain data that can be accessed and altered by all shader handlers during the execution of a program. Each shader type possesses its own context. In a technical view, this context is mapped to the temporary shader registers. Thus, the reading and writing operations are resolved by the shading language compiler. The interface is an important input to the preprocessing step and must be defined in advance.

4 Shader Preprocessing

Given an ordered list of handler prototypes and a handler interface, a language specific preprocessor parses each shader program (*SP*) to determine the particular handlers and the affiliation to the respective prototypes (prototype-handler mapping). Fig. 3 shows the data flow diagram for preprocessing a single shader. In detail, this process can be divided into four steps:

1. Identify handler and prototype from tagged source and qualify the handler name with the name of the shader program it is attached to.
2. Determine the execution mode of the shader handler. Each prototype possesses a default execution mode. It will be used if the shader handler does not specify one.
3. Generate intermediate shader code that can be interpreted by a vendor specific compiler. The signature of each shader handler will be modified with its qualified name to ensure identity at source code level.
4. Store the qualified name, prototype, and execution mode for each shader in a prototype mapping table.

5 Shader Combination

In contrast to other systems, the advantage of our approach lies in the dynamic generation of an uber-shader program at runtime based on the type information for each shader handler. This process will be performed by a shader management system (SMS) and includes the following steps:

1. Concatenation of the intermediate shader sources. Furthermore, each handler is associated with an index into a global *handler invoker table*. This table is a boolean vector that defines the activity state of a shader handler. All will be provided to the controller shader.
2. Generation of the controller shader. It represents the entry point of the shader program.

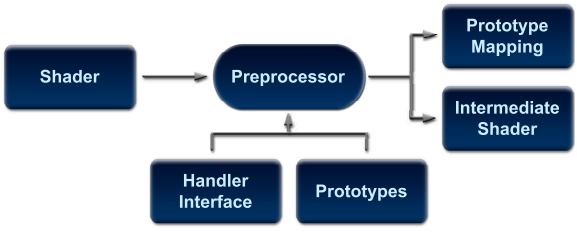

Figure 3: *The preprocessor transforms a shader into an intermediate representation and determines type information of its shader handlers for combination at runtime.*

The maintenance and concatenation of the shader source code is done by the SMS. It controls instances of shader programs in a priority list (*PPL*). The programs are added during the pre-traversal of the scene graph. The *PPL* is used later on to generate the main function of the controller shader. A priority can be assigned to a program that controls its position in the *PPL*. The *controller* is a special shader that invokes each handler if its respective invoker table entry is set to true. It stores the order of handler execution inherently with respect to the prototype list. The following pseudo code shows the creation of the main function:

foreach *prototype* $P \in PL$ **do**
  foreach *shader program* $SP \in PPL$ **do**
    foreach *shader handler* $SH \in SP$ **do**
      if *(prototype(SH) = P)* **do**
        appendIfStatement(SH)

6 Shader Execution

Once the uber-shader source is created, the system must be able to control the execution of the particular shader handler associated with a program. By activating the shader program, the invoker table entries for each handler will be set according to its execution mode and the activity states of all other shader handler will be adapted. The generated controller evaluates the invoker table and executes the respective shader handler in a particular order. The following GLSL code fragment of a generated vertex controller demonstrates the usage of the interface, the invoker table, and the controller:

```
uniform bool vertexHandlerInvokerTable[6];
struct VertexContext {...} Context;
...
void main(void) {
 if(vertexHandlerInvokerTable[0])
  shaderAonInit(Context);
 if(vertexHandlerInvokerTable[1])
  shaderBonTransform(Context);
 if(vertexHandlerInvokerTable[2])
  shaderConTransform(Context);
 if(vertexHandlerInvokerTable[3])
  shaderBonLighting(Context);
 if(vertexHandlerInvokerTable[4])
  shaderConLighting(Context);
 if(vertexHandlerInvokerTable[5])
  shaderAonFinish(Context);
}
```

The evaluation exploits static branching. In contrast to dynamic branching, this method is available in all programmable hardware stages and ensures execution coherence between the parallel shader units.

7 Conclusions and Future Work

We have demonstrated the concept and implementation of an extensible, dynamic approach to combine high-level shader programs. A developer can extend the functionality of shader programs by assigning meta information to the source code. Our implementation is based on GLSL and is integrated into the high-level graphics middleware VRS (http://www.vrs3d.org). The adaptation of this approach for other high-level shading languages and the analysis of performance remain future work.

Acknowledgments

This work has been funded by the German Federal Ministry of Education and Research (BMBF) as part of the Inno-Profile research group '3D Geoinformation' (http://www.3dgi.de).

References

[AV02] ALEX VLACHOS A. T. I.: Designing a game's shader library for current and next generation hardware. In *GDC Game Developers Conference* (2002).

[Ben02] BENDEL S.: First thoughts on designing a shader-driven game engine. In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, Engel W., (Ed.). Wordware, Plano, Texas, 2002.

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream computing on graphics hardware, 2004. submitted to ACM Transactions on Graphics, 2004.

[FW04] FOLKEGÅRD N., WESSLÉN D.: Dynamic code generation for realtime shaders. In *Linköping Electronic Conference Proceedings* (2004).

[Har05] HARGREAVES S.: Generating shaders from hlsl fragments. In *ShaderX3: Advanced rendering with DirectX and OpenG*, Engel W. F., (Ed.). Thomson Learning, 2005.

[Kes06] KESSENICH J.: *The OpenGL Shading Language Language Version: 1.20 Document Revision: 8*, September 2006.

[Mar06] MARK J. KILGARD: *NVIDIA OpenGL Extension Specifications*. Tech. rep., NVIDIA, November 2006.

[McG05] MCGUIRE M.: *The SuperShader*. Shader X4: Advanced Rendering Techniques. 2005, ch. 8.1, pp. 485–498.

[MDTP*04] MCCOOL M., DU TOIT S., POPA T., CHAN B., MOULE K.: Shader Algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM Press, pp. 787–795.

[MSPK06] MCGUIRE M., STATHIS G., PFISTER H., KRISHNAMURTHI S.: Abstract shade trees (preprint). In *Symposium on Interactive 3D Graphics and Games* (March 2006).

[OKS03] OLANO M., KUEHNE B., SIMMONS M.: Automatic shader level of detail. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 7–14.