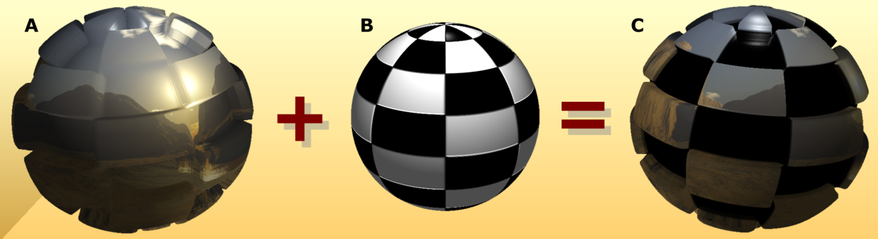
Automated Combination Of Real-Time Shader Programs

Matthias Trapp, Jürgen Döllner
Computer Graphics Systems Group, Hasso Plattner Institute, University Potsdam



Example for the automated combination (C) of displacement, environment (A) and a standard texture mapping shader (B).

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.

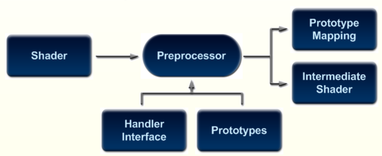
Problem

Due to the restrictions of the current graphics hardware, only a single shader program can be active and replace parts of the fixed function rendering pipeline. With the number of functionality grows also the number of shader permutations. These permutations are hard to manage and maintain. 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.
Reusing such programs saves expenses for repeated optimizations and debugging. This work enables the usage of shaders in a stand-alone way as well as to derive combined variants within a scene graph based rendering system.

Shader Preprocessing

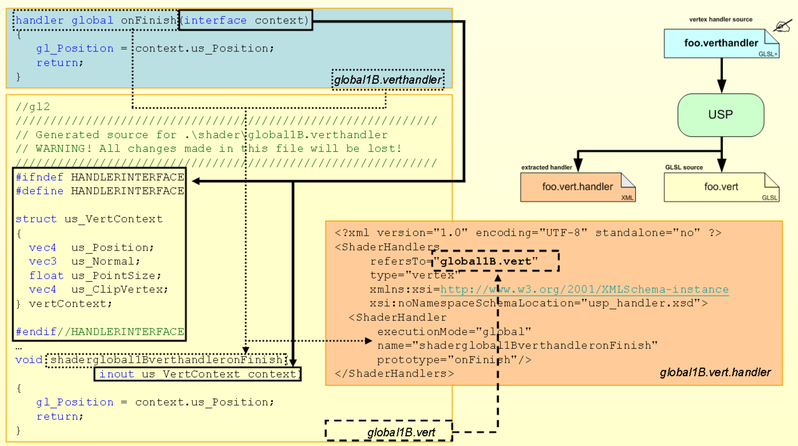
Given an ordered list of handler prototypes and a handler interface, a language specific preprocessor parses each shader program (SP) to determine the particular handlers and the affiliation to the respective prototypes (prototype-handler mapping). Fig. 3 shows the data flow diagram for preprocessing 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.



The preprocessor transforms a tagged shader into an intermediate representation and determines type information its shader handlers.

Example for processing a single vertex shader handler into its intermediate representation and its type information which is necessary for its combination at runtime.



Concept : Divide – Tag - Combine

Divide
Our basic idea is a generic approach for uber-shader construction. It combines 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.

Tag
For later combination, it is necessary to provide additional information 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 that denotes the functionality of a shader. The assignment of such information can be done via tagging the source code by introducing new keywords to a particular shading language (here GLSL):

```
handler [ local| global| optional| ignore] hName ( interface IName)
```

Later, a preprocessor can eliminate this syntactical overhead. The concept can be applied to all shader types (fragment, vertex, and geometry shader). Example for a single vertex handler:

```
handler global onFinish( interface context)
{
  //GLSL source code
  gl_Position =  context.us_Position;
}
```

Combine
To enable a dynamic combination, our approach consists of two separate processing steps:

- A preprocessing step for each shader program analyzes a tagged source code and transforms it into an intermediate representation.
- 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.

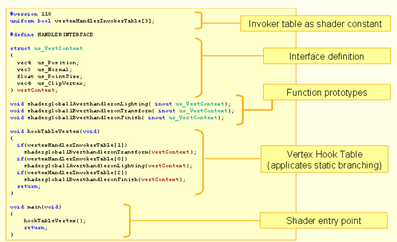
Shader Combination

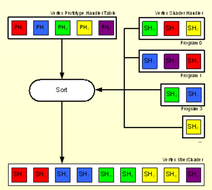
This process will be performed by a shader management system (SMS) and includes the following steps:

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.

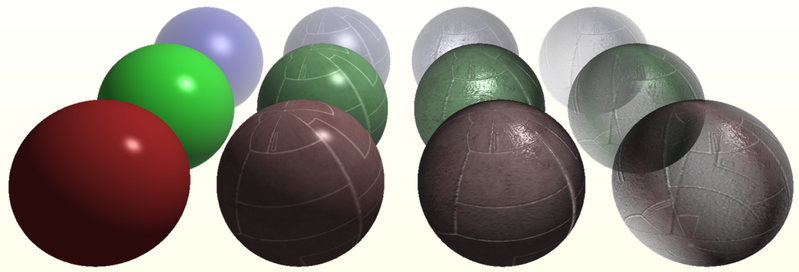
The maintenance and concatenation of the shader source code is done by the SMS. It controls instances of shader programs in a priority list (PPL). The programs are added during the pre-traversal of the scene graph. The PPL is used later on to generate the main function of the controller shader. The controller 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 ∈ PL* **do**
  foreach *shader program SP ∈ PPL* **do**
    foreach *shader handler SH ∈ SP* **do**
      if (*prototype*(*SH*) = *P*) **do**
        appendIfStatement(*SH*)

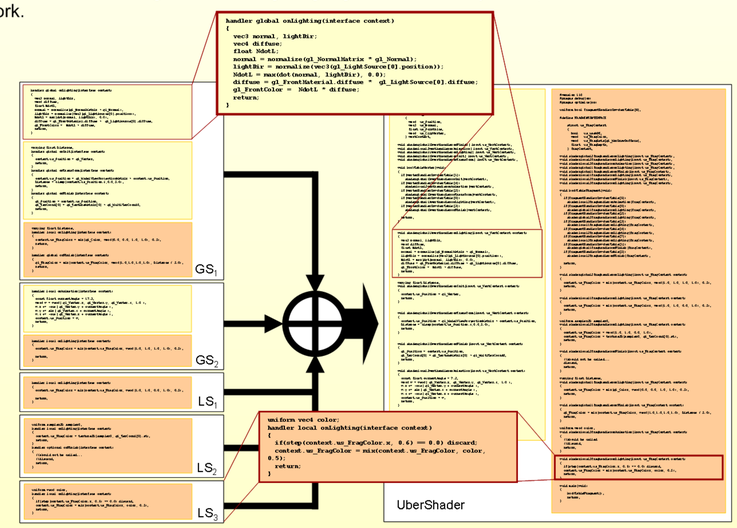




Conclusion & Example


Compositing of different effects rendered by nested shader programs. Phong-shading is combined with texture mapping, normal mapping, and an x-ray shading effect.

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. The adaptation of this approach for other highlevel shading languages and the analysis of performance remain future work.



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 OpenGL*, Engel W. F., (Ed.). Thomson Learning, 2005.
[Kes06] KESSENICH J.: *The OpenGL Shading Language, Language Version*: 1.20 Document Revision: 8, September 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. 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.

Contact:
matthias.trapp@hpi.uni-potsdam.de
juergen.doellner@hpi.uni-potsdam.de

August-Bebel-Str. 88, 14482 Potsdam, Germany

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung