FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Design and Implementation of a Shader Infrastructure and Abstraction Layer

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Michael May
Matrikelnummer 0126194

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Dr.techn. Robert F. Tobler

Wien, 28.09.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Design and Implementation of a Shader Infrastructure and Abstraction Layer

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Visual Computing

by

Michael May
Registration Number 0126194

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr.techn. Robert F. Tobler

Vienna, 28.09.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael May
Arbeitergasse 4/20, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I want to thank my parents for all their support. I would have never come that far without their endless patience.

Additional thanks for all the advice they gave me goes to Robert F. Tobler, Christian Luksch, and Michael Schwärzler.

Kurzfassung

Die Programmierung der Grafikkarte ist wichtiger denn je, aber die Entwicklung für die dafür nötigen Shader Programme und deren Verwaltung ist eine schwierige Aufgabe. Es stellt sich die Frage ob dieser Prozess in die Programmiersprache C# eingebettet und dessen Entwicklungswerkzeuge unterstützend genutzt werden können?

Um dieser Frage auf den Grund zu gehen, wurde in dieser Arbeit ein System entworfen und implementiert um Shader-Programmierung zu abstrahieren und mittels einer *internal Domain Specific Language* (kurz: iDSL) in C# zu integrieren. Das Back-end kann mittels Erweiterungen um beliebige weitere Optimierungen und unterschiedliche Shader-Programm-Dialekte ergänzt werden.

Das implementierte Framework ermöglicht Shaderprogrammierern die Entwicklungswerkzeuge von C# zu nutzen, wie zB. automatische Textvorschläge und Vervollständigungen oder Typ-System-Fehlererkennung im Editor.

Diese Arbeit ist entstanden in Kooperation mit dem Österreichischen Entwicklungsunternehmen VRVis.

Abstract

Programming the GPU is more important than ever, but the organization and development of shader code for the GPU is a difficult task. Can this process be embedded into the high level language C#, gain from the features of its toolchain and enrich shader development?

For this purpose this thesis describes the design and implementation of a framework to abstract and embed shader development into C# with an *internal domain-specific language* (iDSL for short) as front-end and a plug-in system in the back-end to support expandable optimizations and different shader languages as targets.

The implemented framework fits shader development into the C# toolchain, supporting autocompletion, and type error checking in the editor. The system offers good modularity and encourages developing shaders in reusable parts.

This diploma thesis was developed in cooperation with VRVis Research Center in Vienna, Austria.

Contents

CHAPTER 1

Introduction

People have always been telling stories in pictures: First drawings, then photographs and movies. All of those methods take 3d scenarios and put them on a 2d canvas. This transformation is either done by the imagination of an artist or by a light sensitive surface (e.g., film or CCD).

A mathematical approach to transform 3d objects onto a 2d canvas can be done with the help of geometry. Objects are broken down to geometrical primitives (e.g., points, triangles, or general surfaces) in 3d space and the projection to the 2d canvas is calculated. With the advent of the computer, such mathematical abstractions of reality could be automatically converted much faster. Increasing computational power made it possible to use models with increasing detail and the creation of the 2d image became increasingly more realistic. Today, those reality-imitating computer images are used in all different kinds of industries, e.g., movies, interactive applications (e.g., computer games), visualization in hospitals, construction visualizations, and more.

The graphics card is a highly specialized piece of computer hardware that takes data like 3d

Figure 1.1: Depth of field, motion blur, and other effects are presented in this picture with CryENGINE® 3 by Crytec® .

points of objects and generates 2d pictures on the computer screen. It can be programmed with pieces of code called *shaders*. These programs can modify the position of the 3d points that make up a model (e.g., a statue or even rain drops) or define the resulting color on the screen. This can be used to implement different effects like a leather surface, rain drops, or the visible distortions of a motion blur (see Figure 1.1).

Shaders are at the heart of the industry race for faster and more realistic graphics. But their development has some open challenges where best practices are still to be found. This chapter explains the context of this thesis starting with some history, shows the general problems and goals of this thesis and concludes with an overview of its structure.

1.1 Context

1.1.1 A Compact History of the GPU

In the beginning of the 1990s, computer and console games with software-rendered 3D graphics became widely popular, like the first-person shooter DOOM® [Id]. This demand started an arms race of 3D hardware acceleration cards, comparable to the performance race of CPUs (central processing unit). A rule of thumb called *'Moore's law'* describes this ongoing advancement of hardware: "the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years" [Wikb, Moo65] (see Figure 1.2).

Parallel to the hardware efforts, some software 3D graphics APIs were developed. 1992, SGI® (Silicon Graphics Inc.) created *OpenGL*® (Open Graphics Library) based on their proprietary code IrisGL, and Microsoft® introduced *Direct3D*® in 1995 [Wika].

OpenGL® with its extension facility has the edge with hardware developers introducing extensions as soon as they have that feature in hardware, but to support them in a release can take longer. Since Microsoft® started working closer together with manufacturers, those new features are often introduced simultaneously with a new version of Direct3D® .

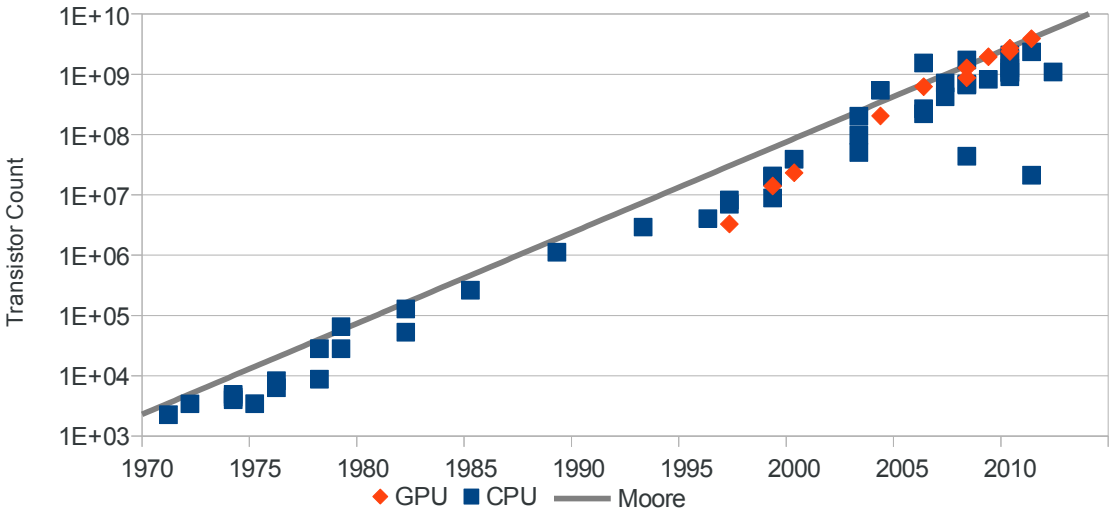


Figure 1.2: This graph shows the increase of transistor counts over time for CPU's and GPU's (Data from [Wikd]) compared to *Moore's Law* as described by Gordon E. Moore in 1965 (see [Wikb, Moo65]).

Nowadays, even Desktop systems like Mac OSX or Microsofts Windows leverage the power of modern GPUs. Tablets and cellphones are used for 3D gaming, and manufacturers try to merge CPU and GPU to make even more powerful systems. Effective programmability of those devices is more important than ever.

1.1.2 Real-Time Rendering

Rendering is the method of generating a 2d picture from 3d models. The amount of data to process and the complexity of the scenes make this only feasible on a computer. For a whole animated movie it takes over hundred thousand of such still images and one image can take hours to process to get the desired degree of realism. To accomplish this amount of computations in feasible time a network of computers is used.

Real-time rendering produces rendered images at interactive rates. Instead of pre-computing still images as for movies, those images have to be computed in a fraction of a second, because the images depend on the interactive input of a user, e.g., in computer games.

The speed of those calculations is measured in FPS (frames-per-second). From 15 fps onwards, it can be called real time, but single images are easily distinguished. Movies are traditionally shown with 24 fps to make the user unaware of single images and perceive fluent movements. At a rate of 72 fps, differences between frames are effectively undetectable [AMH02]. As mentioned above, movies can pre-compute the right amount of images to achieve the desired fps. For real-time rendering, an image has to be calculated in less than 40ms to reach 24 fps.

1.1.3 Hardware Acceleration and Shaders

To make more complex scenes and realistic images possible, hardware acceleration in form of the GPU (graphic processing unit) is used. In the beginning of the 90s, they were tailored to a specific rendering method with configurable algorithmic extensions. The basic method still prevails in form of the graphics pipeline (more details in Section 2.2.1), but has programmable parts now, called *shaders*. Developers can program all different kinds and combinations of effects with shaders, e.g., depth of field or motion blur (see Figure 1.1).

A shader is taken from a string of characters inside the application language or loaded from a text file and sent to the shader compiler of the graphics driver. Communication between the application language and the shader is done with the help of a graphics API (e.g., DirectX® or OpenGL®).

1.2 Motivation

Integrating shaders into an application needs tedious glue code, and deeper manipulations of the shader code during runtime of the application needs string manipulations. Therefore an embedding of shader development into a higher-level application language (e.g., C++ or C#) would be highly beneficial. Instead of defining a shader in a separate shading language, it would be defined with constructs of the application language. Such an API is called an *internal domain-specific language* or *iDSL* (see Section 3.2).

This offers the following benefits:

- Direct influence on the shader program by the application language (no glue code)
- Direct shader program manipulation without string manipulation (e.g., loop unrolling)

- More direct interaction with the specification of textures, attributes, and parameters of the application language
- Use of language constructs of the application language inside the shader
- Use of development tools of the application language

Such a system was implemented by McCool et al. in C++ [MQP02]. Modern applications tend to use younger languages like C# because of their vast standard library, development tools, and features, e.g., a garbage collector and reflection. Therefore, C# was chosen for this thesis to make those features available to shader development.

1.3 Goal, Challenges, and Contribution

The goal of this thesis is to embed the development of shaders into a higher-level programming language according to the iDSL paradigm. To concentrate on those aspects, an existing render system will be used as a basis, but the methods proposed are not tied to that system. The design and implementation presents several challenges:

Developer's view The system should be easy to use for developers already familiar with shader development. *The challenge* is to create an interface that feels familiar to conventional shader programming. The iDSL has some limitations by its dependence on a host language. All constructs of the iDSL to define a shader have to be valid code inside the host language and therefore shaders will look different from those written in a shading language.

Enhanced toolchain Application languages mostly have much better debugging and developing tools than shader languages. *The challenge* is to make those features usable for the shader development process by implementing a good integration in the host system. For example the proposed system provides C#'s autocompletion for all shader constructs.

Hardware shader abstraction A shader-independent framework can support all types of hardware shader and might even produce shaders for a CPU renderer. *The challenge* is to find the lowest common denominator of hardware shaders and implement a plug-in system that can support the creation of different types of shader languages. The proposed system uses the visitor-pattern (see Section 2.5.2) as plug-in system for all operations on the shader tree, e.g., shader creation or optimizations.

Abstract shader representation Shader specifications and algorithms have to be analyzed to find the best way of building an abstract framework for shader development. *The challenge* is to encapsulate all shader functionality with all its language constructs inside the higher-level language. This does not just involve call wrapping, but an interface to create an abstract shader representation. This is done by a tree structure, which is called *shader tree* (see Section 3.3).

The main contribution of this work is the embedding of shader programming in C# as a host language and to show the possibilities of this enriched development process, for example by providing the host's code autocompletion and live error checking.

1.4 Thesis Structure

Chapter 2, Background gives an overview of the state of the art. The description of shading and shade trees should make the problem domain clearer. A detailed look at hardware shaders with the graphics pipeline and an overview of domain-specific languages introduce the technologies this thesis works with. This is followed by a look at further related technologies to show what has been done and also what can be possible with the approach introduced in this thesis. A short introduction of design patterns is given in the end to introduce concepts that are used in a later chapter.

Chapter 3, Design describes concepts based on the ideas introduced in the former chapter and used in the later. The shader tree and its nodes are formulated and the workflow with its components are laid out.

Chapter 4, Implementation introduces the host language, the render system, and how specific features of them are used by the system. HLSL is the example target shader language used during development and therefore described to understand some implementation specifics. The main part of this chapter lays out the implemented components and their features. The last section shows the extensibility of the system for each part of the workflow.

Chapter 5, Example shows an example scene and some of its implementation specifics which was created to learn more about the problem domain and test out the system during development. It shows features of the system and how it handles not yet implemented parts of shader capabilities.

Chapter 6, Evaluation compares the solution to legacy shader code and other shader frameworks in categories like compile time, shader complexity, or feature set.

Chapter 7, Future Work describes additional ideas from previous works or new ideas that came up during development and couldn't be implemented because of lack of time. Design suggestions for the implementation of those feature are given.

Chapter 8, Conclusion looks back on the described system and evaluates how the challenges were met and brings some final thoughts about the system.

CHAPTER 2

Background

This chapter highlights the background of this thesis. The main part is describing the three fundamentals which this thesis builds upon: *shade trees*, *hardware shaders*, and *domain-specific languages*. Further topics of interest are described, e.g., shader level of detail or visual authoring. In the end a short description of design patterns is given which are used in this thesis.

2.1 Shading

Shading in 3D rendering describes the visual representation of a surface in the generated image. This can be a combination of effects from light-sources, shadows, textures, colors, and more. The first rendering systems had fixed models for shading. As more algorithms for different surfaces were developed, a more flexible approach was needed.

Shaders are algorithms for shading given to the rendering system. There are four main approaches:

Declarative Shade trees started the trend of flexible shading algorithms (see Section 2.1.1). It is still used by visual editors (see Section 2.4.4) to empower graphics artists by an easy way to define shaders for their models. Although basic control flow is possible to map visually, declarative models never seem to fully convince programmers.

Procedural Procedural shaders are the industry norm with graphics hardware being programmed by shaders mimicking the syntax of the procedural language C (see Sections 2.1.2 and 2.2).

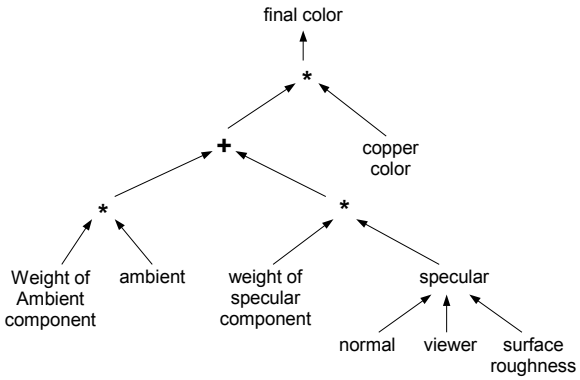
Functional It has been argued that shaders map well on functional languages, with side-effect-free shader stages working parallel over a stream of data, reminding of pure functions over lists. Different projects have implemented functional approaches (see Sections 2.3.1 and 2.3.3). Although functional programming has a long history and highly advanced features, it is not a mainstream programming concept and traditional shader programmer might not see a benefit in mapping their procedural concepts to functional ones.

Object Oriented Extending procedural shader languages with object oriented designs creates a tighter coupling with an object oriented application language, but it relies on the shader compiler to optimize the added complexities [Kuc07, KW09]. Also the introduction of

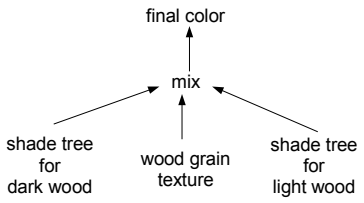
shader interfaces (see Section 2.2.2) introduced object oriented design for dynamic linking natively.

2.1.1 Shade Trees

Robert L. Cook introduced a flexible tree-structured shading model in 1984 [Coo84] to create more complex shading characteristics, which were not possible with traditional fixed models. It describes a directed acyclic graph with each node being an operation like multiplication, dot product, or specular lighting. The leaves of the tree are the inputs and the root is the final color (see Figure 2.1).



Shade tree for copper.



The mix node in a shade tree for wood.

Figure 2.1: These two graphs show shade-tree examples from Cook's paper [Coo84], which describe surfaces as an acyclic tree (the technique also covers the description of light sources and atmospheric effects).

For each type of material a specific tree can be tailored. Besides surfaces Cook also used separate trees to describe light sources and atmospheric effects. A special shade tree language was created to describe a tree (see Listing 1). It lacks any higher control flow, but has some build in nodes and also supports user defined nodes.

```
1 float a=.5, s=.5;
2 float roughness=.1;
3 float intensity;
4 color metal_color = (1,1,1);
5 intensity = a * ambient() + s
6     * specular(normal, viewer, roughness);
7 final_color = intensity * metal_color;
```

Listing 1: This is shade tree for copper (Figure 2.1) written in Cook's shade tree language.

2.1.2 Procedural Shaders

To create more powerful algorithms than it was possible with shade trees, Perlin introduced a *Pixel Stream Editing Language (PSE)* in 1985 [Per85]. This language has similar features as the

programming language C, including control flow structures and function definitions, but also features specific for its graphical domain like vector types and a build in graphical library. Perlin defined images as a 2d array of per-pixel data. The PSE has such an image with arbitrary per-pixel data as input and output and the algorithms are run for each pixel. An example for such data can be seen in Equation (2.1).

$$[surface, point, normal] \rightarrow [red, green, blue] \tag{2.1}$$

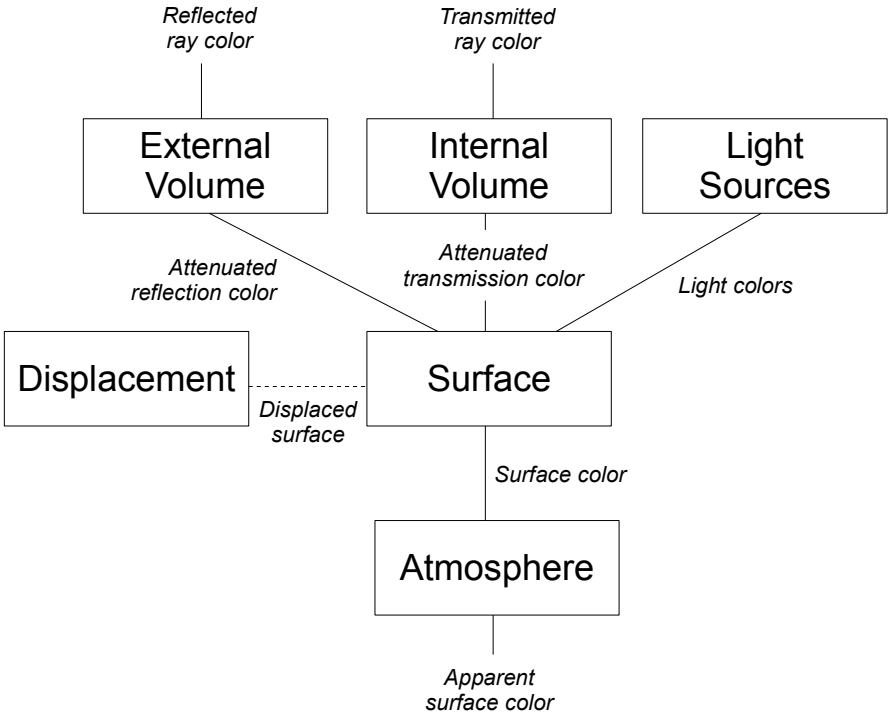


Figure 2.2: The RenderMan® shader evaluation pipeline [HL90] introduced displacement and volume shaders.

Hanrahan and Lawson created a language like the PSE for Pixar's® RenderMan® calling it a *shading language* [HL90]. Inspired by Cook's shade trees, they split the processing in separate shaders for light source, volume, surface, and atmosphere calculations (see Figure 2.2). This shader functions are always called by the render system and never by another shader. Control from outside the render system is only possible via arguments that are passed through to the shader. The RenderMan® shading language is part of the RenderMan® interface which defines an interface between modeling and rendering software and is used by commercial and open source programs.

2.2 Hardware Shaders

In 2001 NVidia® introduced the GeForce® 3. It was the first GPU with *programmable shader parts* [LM]. Until then developer could only configure a fixed-function-pipeline. With shaders,

part of that pipeline could be freely programmed. This was done in an assembler language and was soon adopted by other manufacturers as well. To make developing easier NVidia® and Microsoft® cooperated to create a C like shader language. In 2002 NVidia® released Cg and Microsoft® HLSL for DirectX® 9. The OpenGL® ARB (architectural review board) standardized their higher level shader language GLSL in 2004 for OpenGL® 2.0.

Hardware rendering systems have a streamlined data flow without side effects. This means that calculations in one stage only have an effect on the next stage and don't influence shaders on the same stage. This makes parallel computation highly beneficial. Different Pixels as well as Vertices can be computed at the same time. This is reflected in the graphics pipeline and implemented on hardware with so called stream-processors. With shading languages getting more powerful and more generic, GPUs were started to be used for more generic parallel data processing called *GPGPU (General-Purpose computing on GPUs)*. 2006 NVidia® introduced the C-like programming language CUDA (Compute Unified Device Architecture) to better support the development of non-graphics tasks. Apple® developed OpenCL® (Open Computing Language) for this purpose and it got standardized by the Khronos Group in 2008. In the same year Microsoft introduced their own approach based on HLSL and called it Compute-shaders [Micc].

2.2.1 Graphics Pipeline

The process of creating a 2-dimensional picture out of 3d-data can be highly parallelized. In the most simple picture each pixel can be calculated independently. To utilize this parallelism the process is based on a pipeline concept called the *graphics rendering pipeline* [AMHH08].

The basic conceptual stages of the pipeline are:

- *Application stage* which does e.g., collision detection or animations,
- *Geometry stage* which does transforms, projections, or lighting, and
- *Rasterizer stage* which draws the final image.

Today's GPUs implement the *Geometry* and *Rasterizer stage*. They operate on primitives like polygons or points which are the output of the *Application stage*. OpenGL® and Direct3D® are the API's to work with the stages on the GPU. They expose the same basic pipeline which can be seen in Figure 2.3. The *Input Assembler* processes incoming primitives from the *Application stage* for the *Geometry stage* and the *Rasterizer* is the beginning of the *Rasterizer stage* all the way to the *Output Merger*.

Overview The modern hardware pipeline (Direct3D® 11 [Mica] and OpenGL® 4) consists of programmable parts called shaders and fixed function parts. An overview of the pipeline can be seen in Figure 2.3 with fixed parts as gray rectangles and shaders as blue circles. At least vertex and fragment shader have to be given to render a picture, the others are optional. The shading language to use depends on the API (see Section 2.2.2).

Fixed function parts The *Input Assembler* reads primitive data like points or triangles from buffers and prepares them for the other pipeline stages. The second task it performs is to attach system generated values like vertex-id to the primitives.

The *Rasterizer* converts the vector information of the primitives to fragments/pixels of a raster image. This includes clipping to the view frustum and converting homogeneous clip-space to view-space coordinates of the 2d view-port.

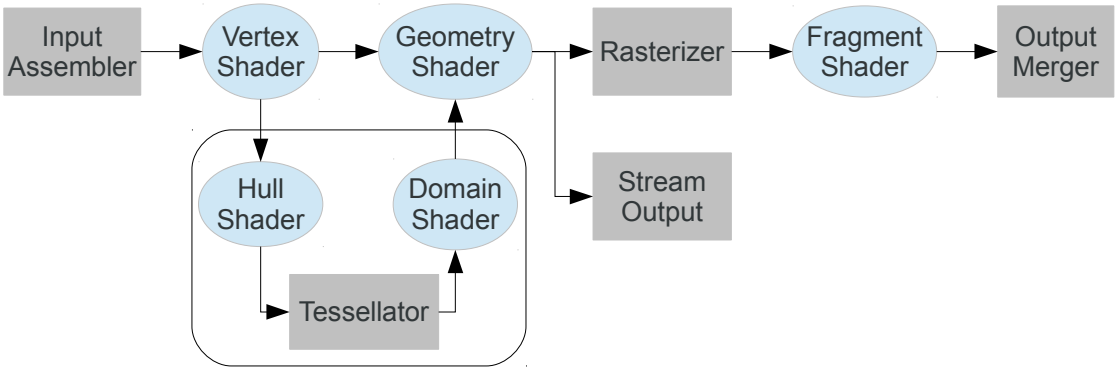


Figure 2.3: This graph represents the render pipeline as of Direct3d® 11 and OpenGL® 4 with the shader stages in blue circles and fixed pipeline parts in gray rectangles.

The *Output Merger* sets the final pixel in the render target, which can be the frame buffer or a texture. It gets the color from the fragment shader and does depth testing and blending.

Stream Output is used if the generated data is not meant to be displayed, but to be read back to the CPU or fed to another pass threw the graphics pipeline. In this configuration no fragment shader is needed.

Vertex shader It runs once per vertex and used for e.g transformations, skinning, morphing, and per-vertex lighting.

Geometry shader It runs once per primitive with additional access to edge-adjacent vertices and can generate new vertices. Algorithms that can be implemented here are dynamic particle systems, point sprite expansion, fur/fin generation, and many more.

Fragment shader It is run for every fragment and calculates the final pixel color and parameters for the output merger. Examples for algorithms implemented here are per-pixel lighting or post-processing effects.

Tessellation The conversion of lower-detail subdivision surfaces to higher-detail primitives is called tessellation. The CPU can handle smaller models and save bandwidth when sending it to the GPU. Other benefits occur like level-of-detail dependent tessellation.

The *Hull Shader* transforms input control points of a lower-detail surface into control points of a patch. In OpenGL® there is one shader per control point. In DirectX® it is defined as two functions: One that is called once per control point to generate the new one and another function called once per patch to calculate patch constants like the tessellation factor.

The *Tessellator* subdivides the domain (quads, triangles, or lines) based on the control points and constants(e.g., tessellation factor) of the hull shader into smaller objects.

The *Domain Shader* is called once per control point created by the tessellator and calculates the vertex position for each.

2.2.2 Shading Languages

Assembly shaders First shaders were written in a basic assembly language. They were introduced in DirectX® 8 and in OpenGL® as extensions. They are now superseded by higher

level languages, but DirectX® still uses a binary representation of the assembly language as intermediate format. This is also possible, but not common with OpenGL® .

Higher level shaders There are three standards for shaders in hardware graphics: Cg from NVidia® , HLSL for Direct3D® , and GLSL for OpenGL® . They all have the same basic functionality, but differ in naming and some higher level language features, e.g., Interfaces in Cg.

The first versions of shader languages had different instructions sets for each shader stage. Later a unified shader model was introduced, where each shader stage shares the same basic language features and only differs in higher functions because of the position in the graphics pipeline.

Cg and HLSL have a lot of similarities based on their identical development roots. They also have more features than GLSL like HLSLs effect files or Cgs Interfaces. HLSL is bound to DirectX® and therefore to operating systems from Microsoft® . GLSL is only dependent on OpenGL® and works on several operating systems. Cg can be used with either DirectX® or OpenGL® and their supported systems.

Dynamic linking Switching between shader features during runtime is done by shader conditional statements or with shader switches. A compromise between those two is dynamic linking. It is only done once per shader permutation and not every shader stage as conditional statements, but also is more modular than specific shaders for shader switches.

An approach to dynamic linking is the *Fragment Linker*. Therefore a shader can be broken apart into smaller fragments. This parts can be compiled once and linked together in different combinations to create different shaders with the help of the fragment linker. Fragments consist of shader functions and the final shader needs one main function to make the right use of them. Global optimizations can not be done at compile time, but at the time of the linking. This feature is not available in DirectX® 10 any more.

In DirectX® 11.2 Microsoft introduced a shader linker that can be separately called from the compiler and the *Function Linking Graph* (FLG). This enables the programmer to compile pieces of shader code in advance like the Fragment Linker. These parts are either executed by and linked with other shader code or connected with the help of the FLG. The FLG is a C++ API that creates a complete shader by combining parts to build a simple shader tree [Micb].

Cg is using an object oriented approach for dynamic linking [Pha04]. *Interfaces* like in C# were introduced to the language. The shader uses instances of an interface to call its methods. Implementation of the used interfaces must be given at compile time, but the linking of an implementation to an instance variable is done during runtime. HLSL followed up on this feature, but uses the *class* instead of the *struct* keyword for implementations.

GLSL offers dynamic linking in form of function pointers and calls it *Subroutines*. Instead of linking interface implementations to instance variables, it links functions to variables. Subroutine types define function headers. A subroutine variable can hold a pointer to a function of a specific subroutine type. A function can implement one or more subroutine types.

Subroutines and *interfaces* both offer the flexibility of runtime conditions without the runtime overhead of shader conditions. They give new possibilities for uber-shader implementations, but still lack higher integration into application languages. Although they can be useful to achieve higher performance, e.g., a shader tree system with conditionals might generate shader code with shader linking if those conditionals have to be evaluated in run-time, but do not depend on in-shader variables.

The *FLG* is an approach to hide away shader languages (e.g., HLSL) and to create shaders with an application language (C++ in this case). Although a basic integration into C++, it is to see, how adaptations of the FLG into other application languages like C# will adopt features of the host language.

2.3 Domain-Specific Languages

A Domain-specific language (DSL) is a programming language of limited expressiveness focused on a particular domain [Fow10]. Shaders are such a DSL with graphics processing as there domain.

For implementing a system with a DSL a semantic model (see Section 2.5.1) is used to separate the domain semantics from the DSL syntax (see Figure 2.4). This benefits the design and testing of such a system.

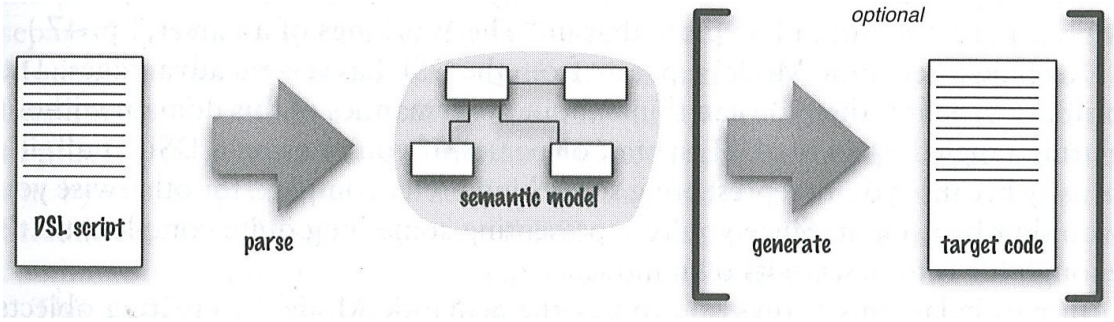


Figure 2.4: The preferred overall architecture of a DSL by Fowler [Fow10] described the separation of domain syntax (DSL script) and semantics (semantic model).

A further categorization of DSLs can be done into external and internal.

2.3.1 External DSLs

They are independent languages with their own toolchain. This tools can be as simple as a parser (e.g., for XML) or as complex as a compiler (e.g., for shader languages). External DSLs are mostly stored in their own file or as strings in another language. Common examples are SQL, XML, shader languages like HLSL, or little languages in the Unix system. Renaissance is a functional approach for a shader language in terms of an external DSL [AR05].

Creating an external DSL gives the most freedom in defining the feature set, like a specific type system tailored to the needs of the domain. The drawback is, that a whole toolchain has to be created.

2.3.2 Internal DSLs

They are embedded into a general-purpose language and also known as *embedded DSLs* or *iDSL*. They are valid code in that language and make use of its development toolchain. Examples for internal DSLs are LINQ in C# and the framework Rails from the language Ruby is seen as a collection of internal DSLs.

Internal DSLs can inherit features from its host language, like the type system or a debugger, but it also has to work inside the hosts limitations. It is fast in prototyping, because no new tools

have to be created. The back-end of an internal DSL has a good scalability. It can be as simple as executing CPU code right away, or more like a compiler with heavy optimizations [BSL+11].

Functional languages like LISP have a long history of internal DSLs and optimizations like 'expression rewriting' seem to come more natural than in other languages. Most main stream programmers might not feel comfortable with the syntax of functional languages, but their concepts slowly find their way in today's more popular languages, e.g., LINQ in C#. LINQ is a framework for list queries which supports different back-ends, e.g., to query SQL databases or internal lists. There is also a project to use it for parallel streaming computations with GPU support [Ana].

2.3.3 Embedded Shader Languages

Embedding shader into an application language has a great benefit to the development process. Programmers don't have to learn another new language and can concentrate on the application language and also all the benefits mentioned for iDSLs apply (see Section 3.2).

Sh library McCool, Qin, and Popa developed a system that integrates shader into C++ and called it Sh [MQP02]. The shader is programmed as a sequence of function calls that generate a parse tree. The tree can perform actions right away and therefore the shaders act like a vector library that performs on the CPU, or it could be further processed to generate shader code for the GPU.

The variables of the embedded shader create an acyclic graph which represents the parse tree. As part of the application language the shader is parsed and type checked at compile time of the application language. To run a shader program it only needs a recursive-descent parser to evaluate the nodes of the parse tree.

Basic functionality like vector multiplication or dot product is supported through parameter overloading. Swizzling, component selection, and write masking is done by overloading the `()` operator. Preprocessor macros are used to make the syntax cleaner. Although they have support for looping, it was not supported in GPU shaders at that time, so the Sh system only could use them, when evaluating on the CPU. Though loops and conditional statement of the application language could be used at any time, they would be unrolled and evaluated when creating the parse tree. The shader iDSL can be put in C++ functions and objects to better organize and facilitate the code.

The Sh library later was extended with the ability to combine or connect existing shaders [MDP+04]. Connecting two shaders creates one shader that feeds the output of the first to the inputs of the second. Combination puts the shaders together into one, which inherits all the inputs and outputs of the combined shaders. To give more precise control to this processes, there are manipulators to e.g., drop in-/outputs or swizzle to rearrange the positions of in-/outputs. Only at the end the type of shader stage is defined and stage inputs/outputs are set.

Vertigo Vertigo embeds shader into the pure functional programming language Haskell [Ell04]. A compiler translates the iDSL shader into efficient shader code for the GPU using partial evaluation and symbolic optimization. An example for the good optimization is the automatic avoidance of multiple normalization of a vector, facilitating expression rewriting.

2.4 Further Topics

Besides making organizing and developing shaders easier, there also has been quiet some work to enrich it with more features.

2.4.1 Computational Frequencies

At different points in a rendering pipeline operations have to be done at a different rate, e.g., processing data on a per-vertex rate is done three times as often on a triangle, than processing at a per-primitive rate. This rates are called *computational frequencies*.

Pixar's® RenderMan® [HL90] introduced two simple rates: `uniform` and `varying`. These rates define constant and dynamic data during shader computations and allow optimization of the code during compilation. These rates were later also adopted in hardware shaders.

The Stanford Real-Time Shading Language (RTSL) [PMT01] extended the rates to *constant, per-primitive-group, per-vertex, and per-fragment* and introduced the concept of *pipeline shaders*.

In this system, shader algorithms are not initially split into stages. Data can be marked for a computational frequency as needed. During compilation the code is split into pipeline stages based on those markers and optimization algorithms.

Renaissance [AR05] implements pipeline-shaders as a pure functional language and Spark [FH11] extends it with an extensible set of rate-qualifiers. Pipeline-shaders can also describe algorithms that reach over several render passes [CNS+01] and is described as the 'Multi-pass Partitioning Problem'.

2.4.2 Shader Level of Detail

A typical speed optimization in computer graphics is *level of detail*. The further away an object is from the viewer, the fewer details can be observed and therefore simplified geometry of the models can be used. This principal can also be applied to shaders. Further away objects can be rendered with simplified shaders (see Figure 2.5).

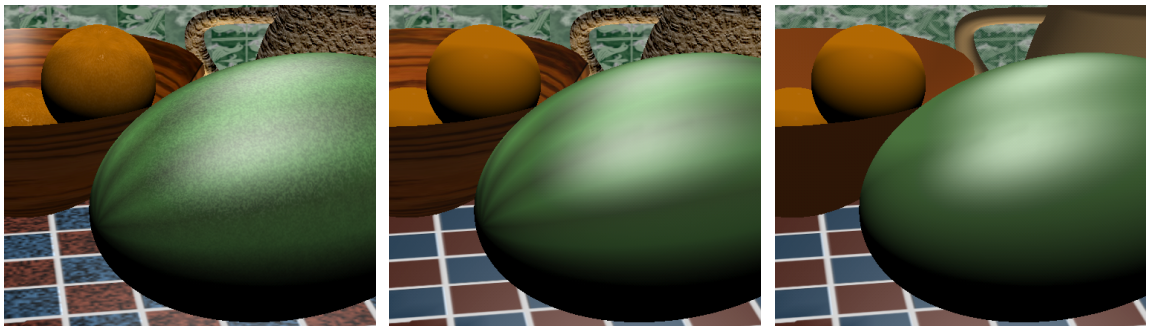


Figure 2.5: Shader level of detail [OKS03] use simpler shader programs for less visible (mostly further away) objects to use less GPU resources at minimal visual impact.

For different rendering effects different algorithms were developed which offer either more realism or more speed. Instead of choosing a compromise for the whole scene, this decision can be made, based on the distance to the viewer. This gives more realism for objects closer

to the viewer and for elements further away faster algorithms can be chosen. There are three possibilities to switch between effect-algorithms of a shader:

- *Pre-compilation* is done by shader switching. Shader with different types of features are compiled and used at a certain level of detail. They can be generated with the pre-processor (e.g., uber shader) or generated out of another shader specification (e.g., shader tree).
- *Pre-linking* uses dynamic linking (see Section 2.2.2) to choose one of the implemented features in the shader.
- *In-shader* branching can be necessary, if the level of detail depends on other shader computations.

All of them have performance advantages over the others in different circumstances, e.g., number of different surfaces and shader length.

Shaders for each of the levels can be handwritten or done with an automated approaches. Olano, Kuehne, and Simmons [OKS03] implemented an automated approach, that simplifies texture lookups from a shader. Either a lookup is replaced with the average color of that texture, or several lookups are merged to one of a combined texture. A cost function estimates the best of those simplifications for different levels. Finally a new shader is created, that decides between the generated level of details in runtime based on the level of detail, which is calculated outside of the shader.

2.4.3 Automatic Shader Generation

Model designers define the materials of their models, but it needs a programmer to make a shader for that material. To make this work easier or even shift shader design to a non-programmer like an artist, feature based code generation is desirable. This means that shader fragments, implementing only certain features, must be put together by the system automatically to generate a complete shader.

This task presents different challenges:

- Decouple features from each other
- Handling of features that spread over several shader stages
- Automatic connections of input and output
- Generate optimized code for shader pipeline

Following solutions have been proposed.

In 'Shader Infrastructure' [CHHE07] a fixed function pipeline is build that generates specialized shaders upon feature requests. The pipeline is a tree of shader fragments which represent different features. It is an uber-shader build in the host language instead of a shader language.

Trapp and Döllner [TD07] partition a shader in parts like lighting, transformation, and others. Different branches of the scenegraph can have their own shader fragments to solve this parts. They are all collected and an uber shader is generated. This creates a scenegraph specific solution and saves costly shader switches.

Folkegård and Wesslén [FrW04] developed shader fragments with pre- and postconditions. The system adds fragments until all preconditions are met. If different branches are possible, an performance optimum path is tried to be selected.

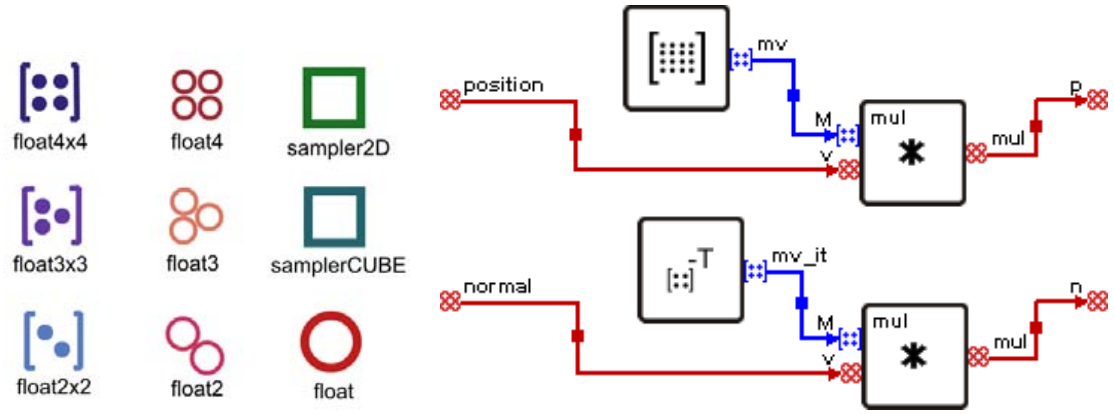
Bauchinger defines in his thesis [Bau07] techniques that can be partly implemented on CPU and GPU. The shaders are connected over Cg-interfaces. The interfaces are defined with their order of execution in the system and can be easily extended. Techniques are chosen based on the requested features.

2.4.4 Visual Authoring

Shader programing is a task, that needs artistic as much as technical skills. The artist that creates 3d models has a specific look in mind, but needs a programmer to formulate this in a shader program. Tools that make it possible to visually create a shader, instead of typing commands, shift this process towards the artist, by making the process easier and in a more familiar way.

The simple principals of Cook's shade-trees (see Section 2.1.1) make it easy to build a visual representation and editor for them. The first implementation of such a system was Abram's and Whitted's 'Building Block Shaders' [AW90]. A more complex system was 'Abstract Shade Trees' [MSPK06] which handles overlapping shader fragments and parameter matching.

A shader extension for the web format X3D was proposed by Goetz, Borau, and Domik [GBD04, GD06]. Their XML based shader language is hardware-shader independent and can be created by a visual editor build in java-swing. They put a lot of thought in the visual presentation of the information in the shader tree, e.g., symbols for each variable type (see Figure 2.6).



(a) Data representations

(b) Example

Figure 2.6: Visual shader editors use graphs to visualize shader programs and symbols for data representation [GBD04].

A visual editor with preview capabilities in each node was developed by Jensen, Francis, and Larsen [JFLC07]. Their editor uses GPU support and therefore previews of parts of their shader tree can be easily created and inserted in their node presentation (see Figure 2.7). It also has extended features like automatic types, geometrical space transformations and code placement. The latter means that only parts of the shader tree are tagged for a specific shader stage, the placement of the rest of the code is determined by their connection to tagged nodes and performance considerations.

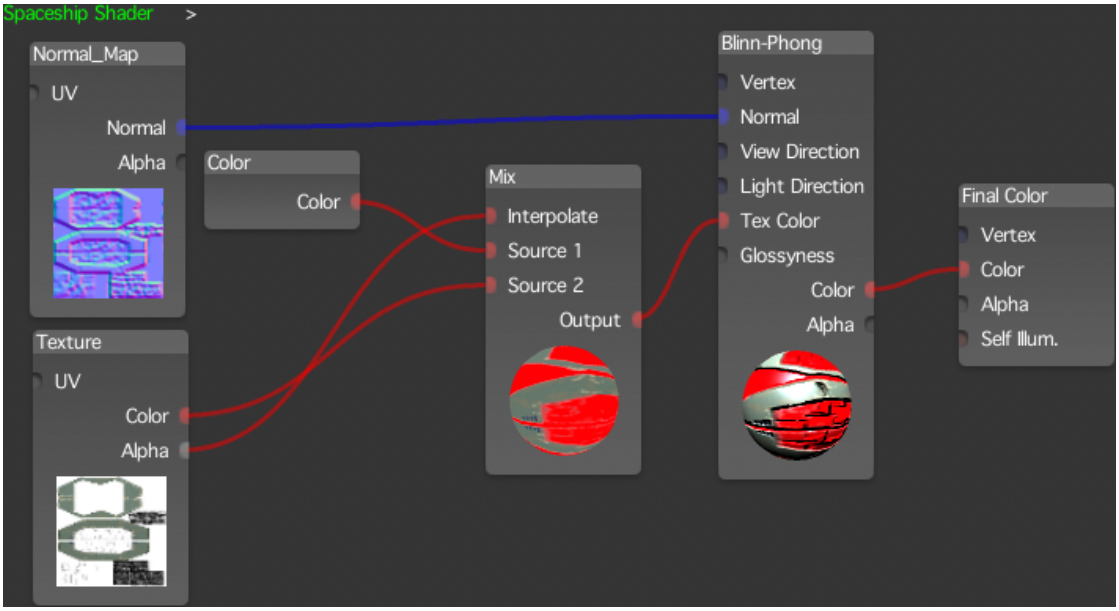


Figure 2.7: Visual shader editors can use the GPU to render live previews as well as the rest of the interface [JFLC07].

Other programs that use visualizations of shader trees, but not for hardware shaders, are Apple's Quartz Composer [Wikc] and different offline rendering systems, like Blender shader editor, Maya Autodesk shader editor, or Softimage XSI shader editor.

2.4.5 Shader Debugger

Shader debugging came a long way from coloring pixel for feedback. Today there are several tools to choose for hardware aware shader debugging. This tools support features that are common for higher level languages for a long time, like stepping through, variable inspection and break points.

NVidia® offers their shader debugger as part of their development platform Nsigth [Nvi] and AMD's complement is part of their GPU PerfStudio [AMD]. Microsoft offers shader debugging as part of its Visual Studio programming environment [Micd]. An open-source solution for GLSL debugging is the tool glslDevil [Kle] (see Figure 2.8).

Shader debugging on such a high level is not an easy task. glslDevil achieves this by hijacking the OpenGL command stream and inserting additional commands in the shader code to pass back debug data [SKE07]. This is done with a library that has the same interface as the OpenGL library and reports to the debug application before it forwards the calls to the real library. This technique can be used without the need of recompilation or source altering of the application software. Shader code can be gathered, when it is sent to the driver for compilation and has to be interpreted by the debugger for command insertion. This makes it possible to read back any data of any shader command, e.g., with NV_transform_feedback. Such modifications have to be done with certain considerations to not alter the behavior of the shader and therefore distort the results. Besides stepping through shader code and reading out variables, it is also possible to collect statistical data and create additional images of fragment debug data.

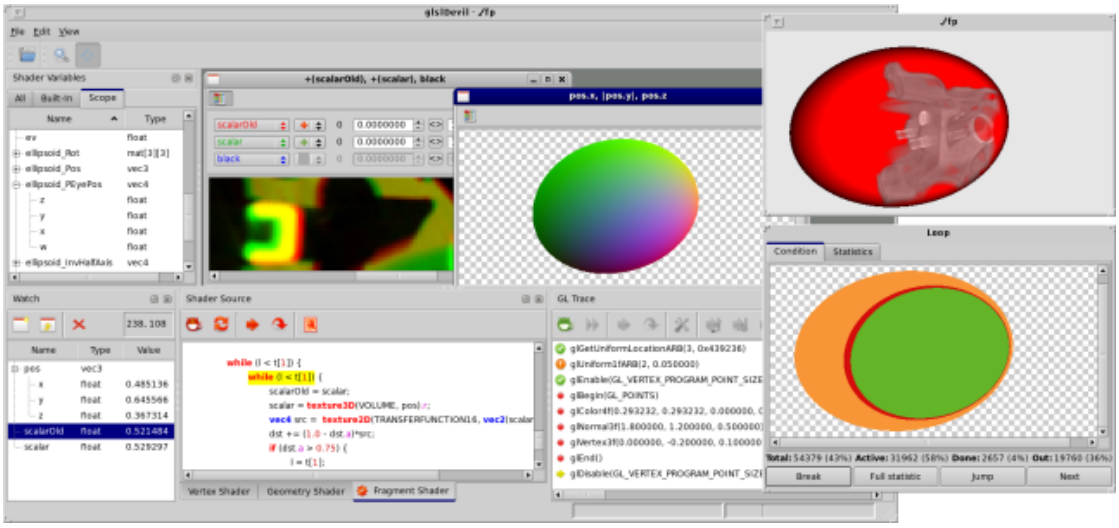


Figure 2.8: The OpenGL GLSL Debugger [Kle, SKE07] supports stepping into the code, partially rendering a shader, and more.

2.5 Design Patterns

Software development strategies that were proven useful have been formulated into *design patterns* for future use and reference. This makes it easier to build upon past experience and helps the dialogue between professionals by expanding their common terminology.

This practice comes from the architectural profession and the first collection for object oriented software development was introduced in *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [GHJV95].

A short introduction to the patterns used in this thesis is given in this section. For further details and research the previous mentioned book is advised.

2.5.1 Semantic Model

This pattern was described by Fowler [Fow10] to separate representation and description of a domain problem. The semantic model represents the subject that was described by a domain specific language. Model designs are diverse and should be based on the purpose. Examples are a state machine, a relational database, or a simple object model.

A model can be accessed in two ways:

- The *Population interface* is used to create the semantic model. This is done by a DSL or by test code to verify functionality independently of the DSL.
- The *Operational interface* acts on a populated semantic model. This can be the execution of code (*interpreter style*) or the generation of code (*compiler style*).

Benefits of this pattern are the dividing of responsibilities into representation and description, the support of multiple DSLs on one semantic model, separated testing of the model, and more.

2.5.2 Visitor

This pattern bundles operations on a set of different objects together. The *visitor* implements the specific operations for each object, while the objects only implement an interface for visitors which executes this object specific code.

Therefore extending the objects with new operations only needs the implementation of these in a new visitor and no change of the objects themselves. Examples for visitor operations are type checking, flow analysis of a syntax tree, or optimizations.

2.6 Summary

This chapter has shown the basic theories and research for this thesis. It builds the ground work for the further design of the framework: techniques to build on and features to build towards. The design patterns should give some basic understanding of terminology and ideas used later on.

CHAPTER 3

Design

This chapter describes the design of the framework based on the previously introduced papers and theories. The base structure to hold the semantics of a shader is the *shader tree* (see Section 3.3). It is a hierarchical tree structure like in Cook's shade trees (see Section 2.2) with the generation of modern hardware shaders in mind. A more theoretical description is given as well as details about the iDSL, the shader tree, and the overall workflow with its components is described. The final section covers further design thoughts that explain some decisions during the development process and parallels to a compiler structure.

3.1 Concept

The main concept has a clear separation of functionality: abstract shader definition (the *iDSL*), abstract shader representation (the *semantic model*), and concrete shader representation (generally called a *view* on the model, see Figure 3.1). Therefore the iDSL can focus on the developer, the semantic model on processing, and different views (e.g., different hardware shader languages) can be supported (an example for all stages can be seen in Figure 3.2).

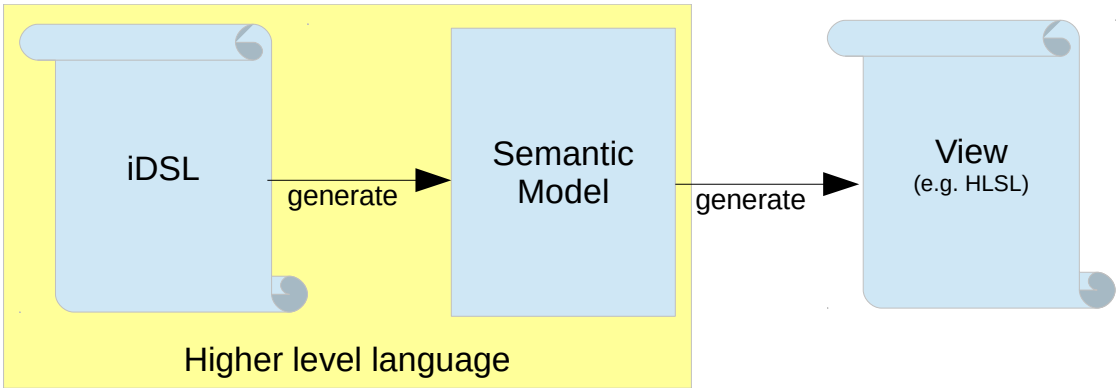


Figure 3.1: The systems concept has a clear separation of functionality into syntax (iDSL), semantics (model), and final representation (view).

All those parts are bound together by the concept of the *shader tree* (see Section 3.3). A shader tree is an abstract representation of a shader (comparable to a parse tree). On an abstract

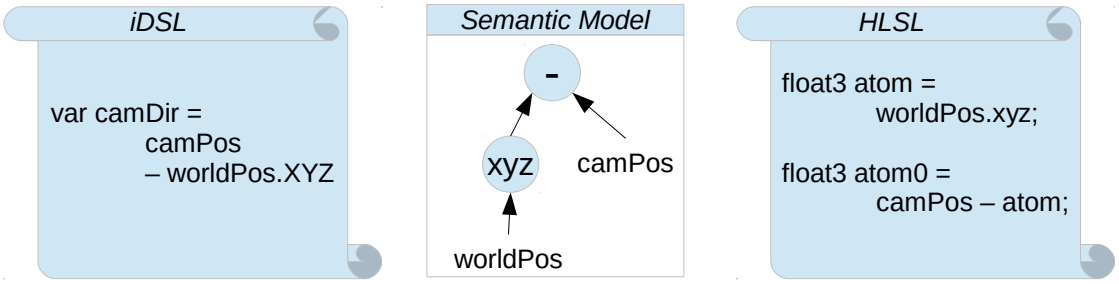


Figure 3.2: Code in the iDSL, it's corresponding semantic model as shader tree, and finally the generated HLSL code is shown in this example.

level the iDSL creates the shader tree, the semantic model holds the shader tree and *views* are created from it. The nodes of the tree represent shader operations and the edges is the information flow between (an example can be seen in Figure 3.3).

iDSL The iDSL is a framework for defining shaders. It is designed for the most convenience of the shader developer in mind (see Listing 2 for an example).

Another goal for the design of this part was the utilization of features of the existing toolchain. The design relies on object-oriented programming to facilitate the use of autocompletion and error checking on the abstraction of the shader type system.

```
1 var camDir = camPos - worldPos.XYZ;
2 var calcLight = CalcLight.Call(normal, camDir, camDir);
3 var finalColor = calcLight.dif + calcLight.spec;
```

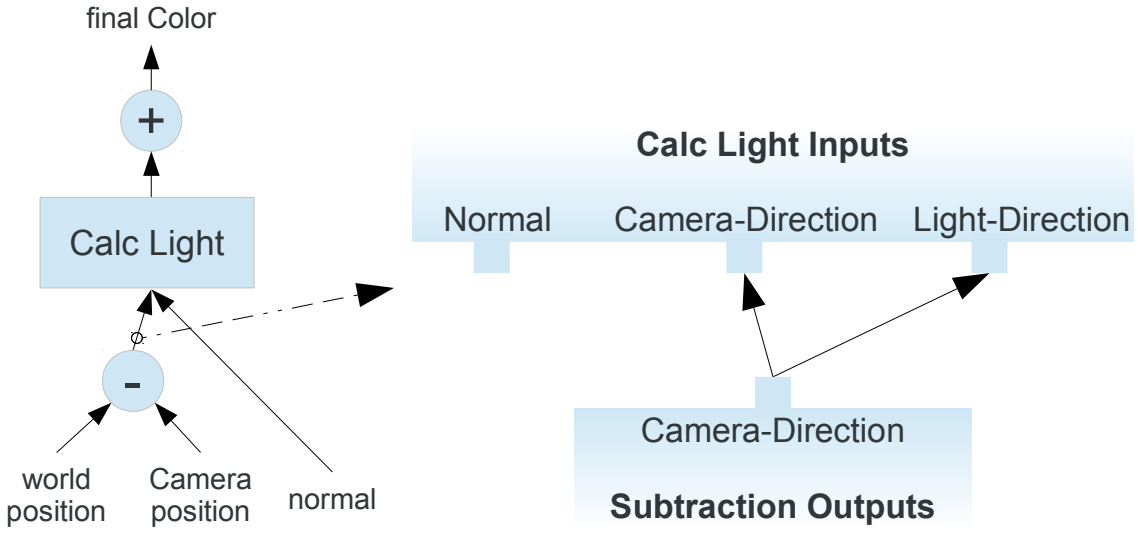
Listing 2: This iDSL example shows simple operations and the call of an iDSL defined shader function in C#.

Semantic model The semantic model is just a simple shader tree implementation. The iDSL already holds all the information the semantic model gets, but is cluttered with code to fit the iDSL well into C#. This makes the iDSL impractical for further processing. Therefore Fowler proposed the semantic-model pattern (see Section 2.5.1), which this semantic model is based on. The pattern defines the necessity of a *population interface* to create the model and an *operation interface* to act on the model.

The population interface is held simple by defining most of the model public. This makes it easy for the iDSL to create the corresponding model.

The operation interface is encapsulated in *visitors* (see Section 2.5.2). This pattern groups together all functionality for a certain process, e.g., shader code generation for a specific shader language or an optimization algorithm. Such a bundle can than be applied on the semantic model. The main advantage of this pattern is the extensibility of the system, e.g., adding another target shader language is just adding a new visitor for this purpose.

An example for a semantic model / shader tree can be seen in Figure 3.3



(a) Shader tree (b) Mapping

Figure 3.3: This Semantic model / shader tree example shows combinations of nodes with edges representing multiple input/output mappings. This example is based on the iDSL example from Listing 2.

View The view is the result of any kind of translation operation on the semantic model. Mostly this would be hardware shader code (see example in Listing 3), but could also just be a diagram of the defined shader structure.

```
1 float3 atom = worldPos.xyz;
2 float3 atom0 = camPos - atom;
3 float CalcLightcall_dif;
4 float CalcLightcall_spec;
5 CalcLight(normal, atom0, atom0,
6       CalcLightcall_dif, CalcLightcall_spec);
7 float atom1 = CalcLightcall_dif + CalcLightcall_spec;
8 float finalColor = atom1;
```

Listing 3: This View / HLSL code is generated from the semantic model / shader tree example (see Figure 3.3), which is based on the iDSL example from Listing 2.

3.2 iDSL

In this section, the fundamentals of the iDSL will be explained with some examples: How to define iDSL variables to interact with C# variables, how to group iDSL code, and how to put together a complete shader in the iDSL. These basics are followed by the explanations of prototyping constructs and C# control flow structures in combination with the iDSL.

Additional insights The iDSL has the shader tree concept at its core (see Section 3.3). Operations are the nodes of the tree, called shader fragments. Input and output variables define

edges.

3.2.1 Defining Variables

The iDSL has a type system that reflects shader types. To make interaction with the host language easier, most iDSL types are mapped to a type from the host language. Many basic types have equivalents like integer or float, some can be mapped to types of the rendering system, like textures, and a few have to be created explicitly, like samplers.

The constructors allow the interaction with native C# types (see Listing 4). This can be used to set default values. Assigning values in C# during runtime sends the new value to the shader on the GPU.

```
1 ShFloat test1 = 47f;
2 ShFloat3 test2 = new[] { 45f, 3f, 90f };
3 ShTexture2D test3 = new Texture("testimage.jpg");
4 ShSampler test4 = new ShSampler()
5           { Type = TextType.D2, Anisotropic = true };
```

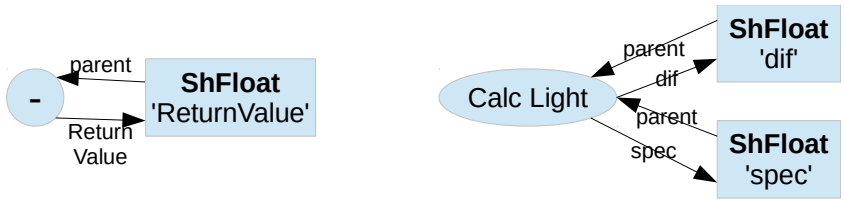
Listing 4: Initialization examples of iDSL variables interacting with natural C# types is shown here.

The following types are implemented:

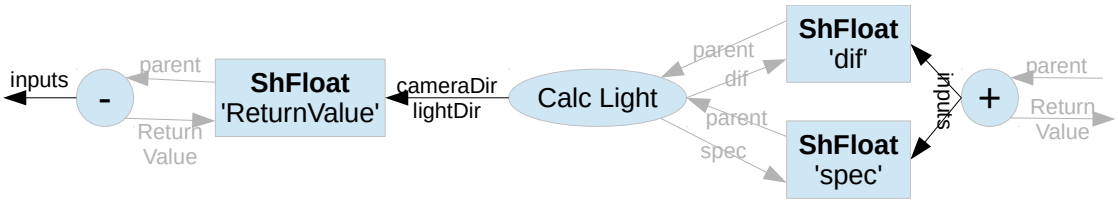
- ShBool
- ShInt, ShUint, ShFloat, ShDouble
- [ShInt, ShFloat, ShDouble][2, 3, 4]
- [ShInt, ShFloat, ShDouble][2x2, 2x3, 3x3, 3x4, 4x4]
- ShArray<T>
- ShTexture[2d,3d,Cube]
- ShSampler

Additional insights Each variable class of the iDSL type system shares some basic attributes:

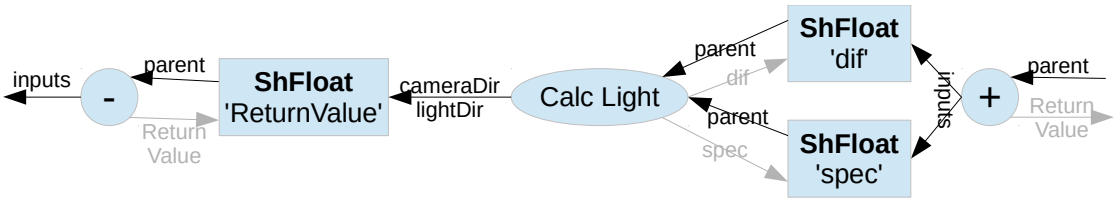
- **Parent** Each output variable knows the fragment it belongs to (see Figure 3.4a). Input variables are initialized by assigning output variables of other fragments. This connects the fragments together (see Figure 3.4b) and is used to parse the iDSL to create the semantic model (see Figure 3.4c).
- **Name** Some fragments (e.g., `ShGroup`) initialize this with a user defined name, others are auto-generated.
- **Default_value** iDSL types that have a corresponding C# type can be set to a constant value which is saved in the `Default_value` property.



(a) On creating an instance of an iDSL fragment, it initializes its output variables by declaring itself as the parent and giving it the name it possesses in the C# code if possible (through *reflection*). Instances are created every time a fragment is used in an iDSL shader.



(b) While building a shader in the iDSL, output variables of fragments are assigned to input variables of other fragments.



(c) The iDSL is traversed from the outputs towards the inputs to create the semantic model.

Figure 3.4: The life-cycle of an iDSL variable includes initialization by its parent fragment, assigning it to an input of another fragment to connect the fragments and traversing it to create the semantic model. This diagrams show internal procedures based on the '-' operator and the 'CalcLight' fragment of the code example in Listing 2.

3.2.2 Basic Operations

Some basic functions are implemented with operator overloading or class methods (see Listing 5). This gives more of the feeling of a real language, than just a simple API.

3.2.3 Grouping

Standard C# functions can be used to bundle iDSL code, but will be in-lined during traversal. For a more modular system, the `ShGroup` class was introduced to the iDSL (see Listing 6).

An *iDSL group* is a class derived from the `ShGroup` class. It is translated to a group fragment in the semantic model, containing all the grouped fragments. Therefore the grouping information is not lost and can be translated to a shader function, instead of being in-lined.

The outputs of an iDSL group are class fields that all have to be declared public. The bundled iDSL code has to be in a class method called `Call`. Inside this method, an instance of the group has to be created to initialize and access the output variables. The input variables have to be registered at the end with the method `InitInputs` for the generation of the semantic model.

The iDSL only uses classes to bundle code and therefore to abstract shader functions. Other systems use them for more complex tasks [Kuc07, KW09], but for this system it was kept

```
var test10 = test1 * test2[1];
var test11 = test2.z - test1;
var test12 = test2.Normalize();
var test13 = test2.Dot(otherFloat3);
var test14 = test3.Sample(test4, test2.xy);
```

Listing 5: Operator and method examples on variables implemented with overloading or class methods is shown here.

```
class ShaderGroup : ShGroup {
    // outputs
    public ShFloat3 finalColor;

    // call method with fragment inputs
    public static ShaderGroup Call(ShFloat4 worldPos,
                    ShFloat3 camPos, ShFloat3 normal)
    {
        // create fragment instance
        var group = new ShaderGroup();

        // define shader
        var camDir = camPos - worldPos.XYZ;
        var calcLight = CalcLight.Call(normal, camDir, camDir);
        group.finalColor = calcLight.dif + calcLight.spec;

        // register inputs and return the ShaderGroup instance
        return group.InitInputs(worldPos, camPos, normal);
    }
}
```

Listing 6: This simple iDSL *group* example encapsulates the shader code from the example in Listing 2. It demonstrates the definition of output variables, creating of an instance, and initialization of the input variables. Using an iDSL group should give the impression of a native function call, but C# doesn't allow overloading parentheses, therefore the static method `Call` is used.

deliberately simple to support coding convenience without diverging too much from classical paradigms.

The big drawback of this method is coding overhead, but it has several benefits:

Extracting variable names C# can extract data of classes like its name, class fields or class method names, and the names of their attributes during runtime by a method called *reflection*. Without extra user effort, this information is collected by the iDSL for the semantic model. Therefore, a function in the generated shader code can have the same names as the iDSL group. If debugging generated shader code is necessary, this makes it easier to map generated code back

to the iDSL (further explained in Section 4.7.2). This can be seen in the example in Listing 13.

Grouping by inheritance A *group* base class can be used to bundle default inputs/outputs with default values and different iDSL code as a basic template or library for further groups.

Instance marking Another debugging method is to name an instance of an iDSL group. This name can be added to the created variables, which hold the return values of the function in the generated shader code corresponding to the group. This can make it easier to distinguish those calls and map them back to the iDSL.

3.2.4 Shader Stages and Putting It All Together

A complete iDSL shader effect is based on the class `ShEffect` (see Listing 7). The `Link` method registers a vertex and pixel shader with the defined effect after connecting them together. Inputs and outputs can be freely defined or the defaults can be used.

Some iDSL fragments (e.g., *groups*) use a `Call` method to simulate the calling of this fragment as if it would be a function. C# doesn't allow the parentheses operator to be overloaded, which would be a more intuitive way to simulate a function call.

A fragment with only one output variable should use this as the return value for the `Call` method. For fragments with multiple output values, the `Call` method returns the fragment itself and the outputs can be accessed through class fields.

```
1  class SimpleShader : ShEffect
2  {
3      public Globals Global = new Globals();
4      public VertexInputs VertexInput = new VertexInputs();
5      public VertexOutputs VertexOutput = new VertexOutputs();
6      public PixelOutputs PixelOutput = new PixelOutputs();
7      public override void Link()
8      {
9          var vShader = vertexShader
10             .Call(VertexInput.Positions, VertexInput.Normals,
11             Global.ModelViewProjTrafo);
12         VertexOutput.Position = vShader.ReturnValue;
13
14         var pShader = pixelShader.Call(vShader.outNormals);
15         PixelOutput.ImageOutput = pShader.ReturnValue;
16
17         Init(vShader, pShader);
18     }
19 }
```

Listing 7: This simple iDSL effect example uses defaults for globals, inputs, and outputs. The `Link` method combines the shader stages (iDSL groups) and registers them with the `SimpleShader` by calling `Init`.

Additional insights In the current system the public members are used to link the outputs to the fragment by setting their `Parent` field (see Section 3.2.1). Probably a more intuitive way would be the use of `out` attributes of the `Call` method and initializing them similar to the input variables.

3.2.5 Prototyping Constructs

For the convenience of fast prototyping based on existing shader code, the iDSL has two fragments that introduce legacy shader code. This should only be used for prototyping because it makes an iDSL shader dependent of one specific shader language.

`ShExpression` is used for short code snippets, typical one liners with only one return value (see Listing 8). This doesn't need a class declaration, but only a method call with the legacy shader code and the inputs as parameters.

```
1 var output = ShExpression<ShFloat4>
2     .Call("mul(_0, _1)", input1, input2);
```

Listing 8: An iDSL expression is used for short legacy shader-code snippets with one return value.

`ShFunction` is used for longer legacy code with multiple outputs (see Listing 9). This needs a class declaration not unfamiliar to the iDSL group. But the `Call` method only needs to create an instance with the shader legacy code and the inputs as parameters. The legacy shader code is typically declared inside the class as a field, but doesn't have to be.

3.2.6 C# Control Structures in the iDSL

C# loop and branch constructs are two features that come for free to the iDSL without additional implementation costs (see example in Listing 10). This includes `for-loop`, `while-loop`, `if-then-else`, `?:`, and others. But it has to be kept in mind that they are all evaluated when the semantic model is created out of the iDSL. So loops will be unrolled and branches will be decided with the information at hand. A separate set of iDSL constructs has to be created to be able to define such functionality in the iDSL that create corresponding shader code that can be evaluated on the GPU (see Section 7.3). A workaround is to use an iDSL function fragment and encapsulate this shader functionality in there.

C# functions can also be used right away, but will also be evaluated when the semantic model is created (see simple example in Listing 11). This gives modularity in the iDSL, but takes away optimization possibilities in the semantic model. The iDSL group fragments supports such optimizations (see Section 3.2.3).

3.3 Shader Tree

The most fundamental structure of the whole design is the shader tree. It is the abstract representation of a shader. The semantic model is an implementation of the shader tree with the purpose of further processing. The iDSL is used to define a shader and generate the semantic model, therefore it has to hold the same information and can be seen as an implementation of

```
class CalcLight : ShFunction {
    // outputs
    public ShFloat dif = ShFloat.Default;
    public ShFloat spec = ShFloat.Default;

    public static CalcLight Call(
        // inputs
        ShFloat3 normal, cameraDir, lightDir,
        ShFloat shininess,
        ShBool doubleSidedLightingEnabled)
    {
        return CreateInstance<CalcLight>(
            Tokens.HLSL, HLSLFuncCode,
            normal, cameraDir, lightDir,
            shininess, doubleSidedLightingEnabled);
    }
    private static readonly string HLSLFuncCode =
      @"// calculate factor: lambert diffuse
        dif = doubleSidedLightingEnabled
            ? (abs(dot(normal, lightDir)))
            : (max(0.000001f, dot(normal, lightDir)));
        float3 r = -reflect(lightDir, normal);

        // calculate factor: phong specular
        spec = doubleSidedLightingEnabled
            ? (abs(dot(r, cameraDir)))
            : (max(0.000001f, dot(r, cameraDir)));
        spec = shininess > 0.000001f ? pow(spec, shininess) : 1;
      ";
}
```

Listing 9: The iDSL function introduces longer legacy shader code with multiple output values.

the shader tree itself. Because of this synergy the shader-tree structure is visible in the iDSL and the semantic model, as can be seen in Table 3.1.

The shader-tree structure is based on shade trees by Robert L. Cook (see Section 2.1.1). His design is a good visualization of the information flow of material shading. The tree could be described by a shade-tree language, which already had the characteristics of a programming language. This language looked like simple C-code and was the base of modern shader languages. For this thesis the idea of the shader code as a tree of operations was taken from Cook's shade tree. Instead of focusing on material shading, this thesis incorporates the pipeline model of modern GPU hardware.

The shader tree is a *rooted directed tree* with the information going towards the root (see Figure 3.3a). Each edge is connected with a source node and a target node. One node has been designated as the root and the direction of the edges goes from the leaves towards this root. Any two vertices are connected with exactly one simple path, i.e., there are no cycles.

```
ShArray<ShFloat> testArray;
ShFloat testFloat;
for (int i = 0; i < array.GetSize(); i++) {
    testFloat += testArray[i];
}
```

Listing 10: C# loops can be used with the iDSL, but are unrolled when the semantic model is created.

```
ShFloat MyFunc(ShFloat input) {
    return input;
}
```

Listing 11: C# functions can be used to group iDSL code, but it is evaluated when the semantic model is created.

Shader tree	iDSL	Semantic model	legacy code
node	`ShFragment`	`Fragment`	
- *atom*	- `ShAtom`	- `Atom`	code
- *group*	- `ShGroup`	- `Group`	function
- *expression*	- `ShExpression`	- `Expression`	code string
- *function*	- `ShFunction`	- `Function`	function string

Table 3.1: The shader tree with its four types of nodes is the basic concept of the design. Therefore the iDSL and semantic model mimic this structure, because they have to hold the same information. In the legacy code the *atoms* and *expressions* become code snippets, while *groups* and *functions* become functions. While *atoms* (see Listing 5) and *groups* (see Listing 6) are defined inside the system, *expressions* (see Listing 8) and *functions* (see Listing 9) are based on strings of legacy code defined during shader development.

Each node has inputs and outputs, and each edge has mappings between the source's outputs and the destination's inputs (see Figure 3.3b).

Shader tree nodes Shaders are broken down into modules of basic functionality. The nodes of the shader tree are either such modules or a group of them and they are all together called *shader fragments*. There are four different types: *group*, *atom*, *expression*, and *function* (see Figure 3.5). All fragments have inputs and outputs. Each input or output has its own name to be distinguished from each other, but it is possible for an input and an output to have the same name.

Group fragments encapsulate a shader tree, so they can be sub-trees of other shader trees. They collect the connection- and mapping informations of fragments. Small groups describe a specific functionality and will be used in bigger groups to finally form a complete shader, e.g., Figure 3.3a shows the hierarchy stored in a group fragment, where 'Calc Light' might describe another group fragment.

Atom fragments describe all basic functionality that is present in all types of shader languages, e.g., multiplication or dot product.

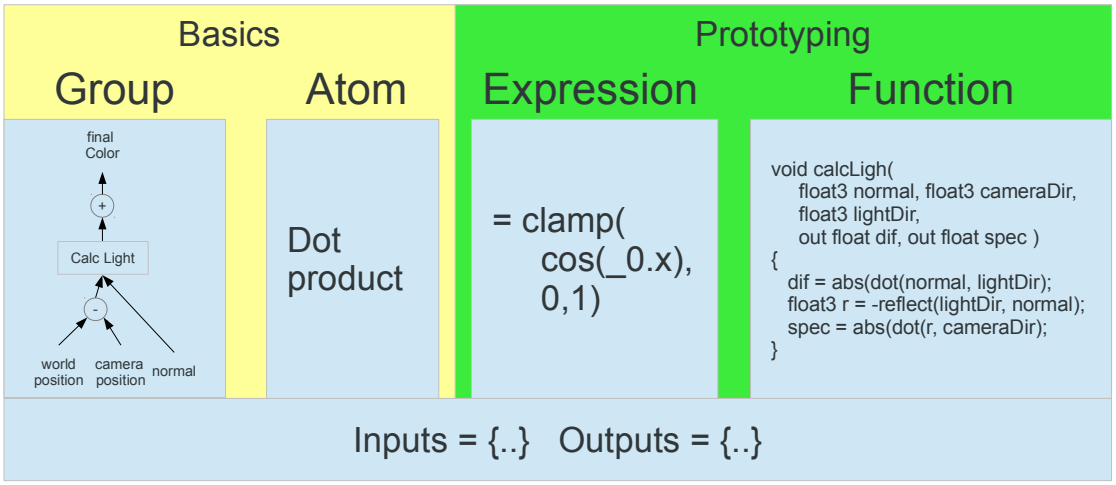


Figure 3.5: Four types of fragments are split into two groups: *Basic nodes* are used for target-independent development and *prototyping nodes* to test existing shader code in a specific target language. Examples for usage are given with each fragment.

Prototyping fragments introduce shader-language specific code. This makes it possible to easily reuse existing code snippets, but limits the use of shader language back-ends. Another advantage is the use of features that are not present in all languages and therefore are not suitable as an atom fragment. There are two of this kind: *expression* and *function*.

Expressions have small snippets of code with one output (the return value). They are good for prototyping with a light syntax.

A *function* fragment represents a complete function and might even be auto-generated from parsing the function header. It has a greater writing overhead than an expression, but can handle multiple outputs.

3.4 Components and Workflow

The system has three major components with a straightforward workflow: The developer specifies a shader in native C# code with the help of the *iDSL*. This is converted to a *semantic model*, where the shader tree can be checked for errors, optimized, or processed otherwise. Finally, it is used by the *code generator* to create shader code for a specific shader language and sent to the shader compiler (see Figure 3.6).

iDSL The main goal of the iDSL is supporting the programmer in defining the shader tree. Higher-level language constructs like object-oriented programming help to construct the shader trees, while debugging tools and the type system support finding errors. An example can be seen in Listing 12.

iDSL fragments are connected to each other by their inputs and outputs (see Section 3.2.1). This describes a tree similar to the shader tree, except the connections point in the opposite direction (away from outputs towards the inputs).

To create a semantic model, this iDSL tree is parsed starting from the outputs (see Figure 3.4c). Each iDSL fragment creates a corresponding semantic-model fragment. iDSL groups collect connection informations from the inputs/outputs and create a semantic-model group frag-

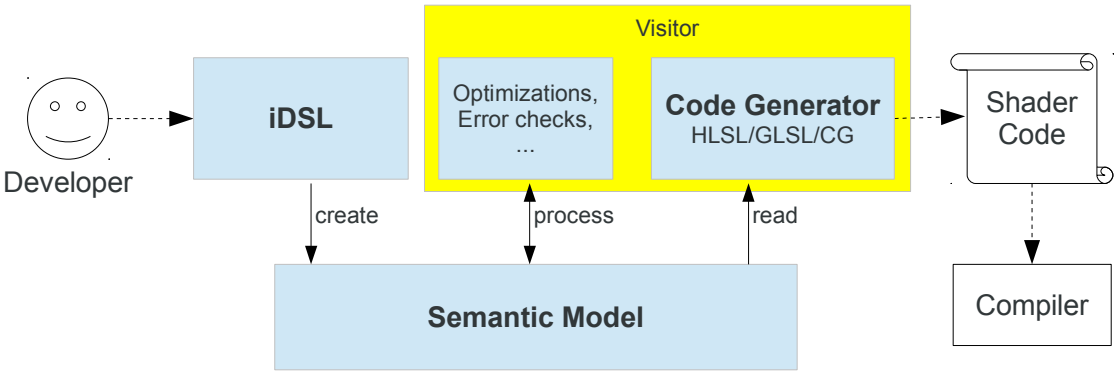


Figure 3.6: This graph shows components and workflow with the basic concept (seeSection 3.1) extended by the *visitor pattern* (see Section 2.5.2) for processing of the *semantic model*.

.

ment with this information. While fragment connection information in the iDSL is held by outputs and inputs, the semantic model saves this in its groups.

Semantic model The semantic model is generated from the shader tree defined by the programmer in the iDSL. It is independent of the other parts of the system, especially the specific shader language of the code generator. Besides decoupling of the components, it is also the base for *general processing*. This can be error handling like finding infinite loops in the shader tree, or optimizations like eliminating code duplication.

iDSL and *semantic model* both are a representation of the shader tree and therefore share the same fragment structure. (Implementation specifics can be found in Section 4.5)

Code generator For each shader language that should be supported, a code generator has to be written. It handles all the specifics of its targeted language and creates shaders based on the semantic model. It also does shader-language specific optimizations, if they are not already covered by the shader compiler. (An example of generated HLSL code can be seen in Listing 13. Implementation specifics can be found in Section 4.6)

3.5 Further Design Thoughts

This section goes into further details of design decisions and ideas of further improvements or alternate routes.

3.5.1 Design

The design of splitting functionality into iDSL, semantic model, and view has one drawback:

Coding overhead Most features will have the need of alteration in all three parts of the design. This can be small, like adding a simple type like Integers, which only needs a few lines. Control-flow constructs like loops, on the other hand, will need more consideration and more code in all parts to implement (see Section 4.8).

But the benefits outweigh this drawback:

```
class TestGroup : ShGroup
{   // outputs
    public ShFloat finalColor;

    public static TestGroup Call(
        // inputs
        ShFloat3 camPos,
        ShFloat4 worldPos,
        ShFloat3 normal
        )
    {
        var group = new TestGroup();

        var camDir = camPos - worldPos.XYZ;
        var calcLight = CalcLight.Call(normal, camDir, camDir);
        group.finalColor = calcLight.dif + calcLight.spec;

        return group.InitInputs(camPos, worldPos, normal);
    }
}
```

Listing 12: This iDSL code shows an implementation of the shader tree example from Figure 3.3. It is implemented as a group fragment and uses a not further specified fragment CalcLight, which might be another group or function. The resulting shader code can be seen in Listing 13. The specifics of the group implementation can be found in Section 4.4.

Support of different iDSL interfaces Implementing a new iDSL interface in the host language is only a change of the iDSL itself. The semantic model and the visitors don't have to be changed. Different input methods can therefore be easily experimented with. For programmers who are more used to functional programming, an iDSL with more functional approaches can be implemented and offered in parallel to other iDSLs. C# already offers functional approaches like the LINQ framework but can also easily cooperate with functional languages that are also based on the Common Language Runtime, like F#.

Host language independence To make the system not only interface independent (can be done by different iDSL implementations), but also host language independent, the model and the visitors could be compiled as a library.

Therefore, iDSLs could be created in different host languages using the library's API to populate the model and use the visitors. To keep the modularity of the visitors, they could be loaded with a module system and therefore would be able to be programmed in a different programming language as well.

However, this approach has a big disadvantage if a new feature is being introduced that needs alterations in all parts of the system. Different programming languages of the library, visitor modules, and application make this more difficult, and if parts are only available in binary form, this task would even be impossible.

```
void TestGroup (float3 camPos, float4 worldPos, float3 normal,
     out float finalColor)
{
  float3 atom = worldPos.xyz;
  float3 atom0 = camPos - atom;
  float CalcLightcall_dif;
  float CalcLightcall_spec;
  CalcLight(normal, atom0, atom0,
        CalcLightcall_dif, CalcLightcall_spec);
  float atom1 = CalcLightcall_dif + CalcLightcall_spec;
  finalColor = atom1;
}
```

Listing 13: This is the HLSL code resulting from the iDSL implementation in Listing 12 of the shader tree example from Figure 3.3. The group fragment is packed inside a HLSL function. Names of input and output parameters of the group are automatically used to name corresponding variables in the generated code.

Simple extensibility Implementing a new shader language as target only needs a new view. With the visitor-style plug-in system, this means implementing a new visitor. Even optimizations can be implemented with only a new visitor (see Section 4.8.3).

3.5.2 Comparison to Compilers

The described shader-tree system bears similarities to a compiler framework (see Figure 3.7). The following section describes parts where they diverge and parts where a shader-tree system can benefit from the knowledge of compiler developers.

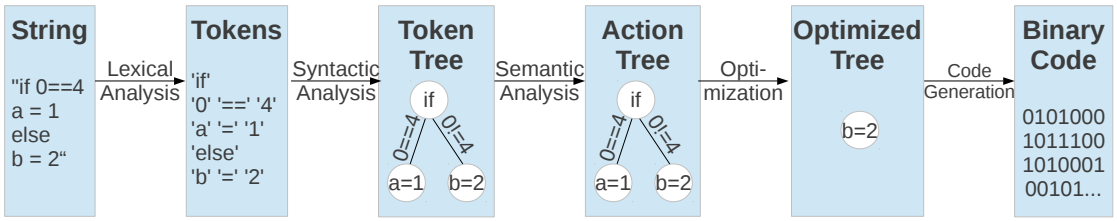


Figure 3.7: The Compiler pipeline transforms code as a string of characters to executable binary code.

Lexical and syntactic analysis The lexical analysis converts sequences of characters into tokens. The syntactic analysis sorts those tokens into a token tree (also called parse tree) based on a grammar.

The C# compiler does lexical and syntactic analysis on the iDSL code, but this is not necessary for the shader code defined by the iDSL. The semantic model nodes/tokens are created at runtime by the iDSL fragments and the grammar is already part of the system. Even the iDSL itself already describes a shader tree made of tokens similar to a parse tree.

Only an eDSL would need a separate lexical analyzer.

Semantic analysis Syntax-directed translation is a compiler method for semantic analysis by attaching an action to a grammar. This is similar to a visitor executing actions on the shader tree nodes of the semantic model.

Code generation This is the core functionality of a compiler to create binary code to be executed. This is also present and a main focus of the shader-tree system, although the generated output might differ. Mostly shader code will be generated, but it is also possible to execute the semantic model right away in a visitor.

Code optimization Visitors are not only meant to generate shader code or execute the semantic model, but also to be used for optimizing the semantic model. This is probably the biggest topic where this system can profit from compiler developers. The design was created with this in mind, but no optimizing visitors were implemented.

3.5.3 Functionality Duplication and Meta Programming

To make the iDSL as powerful as a native shader language, all of its functionality has to be duplicated in the iDSL. This means a lot of initial code writing. There is a lot of similar functionality in such a language, e.g., basic algebraic functionality on different types. These similarities only have to be implemented once in the iDSL with the help of meta programming.

Meta programming Meta programming is writing code that manipulates other source code. A common method of meta programming are templates from C++ or generics in C#. These are powerful tools that are even used to extend their own language, e.g., the BOOST library in C++, but tend to be difficult to debug.

Code generation Therefore a different approach of meta programming was used for this problem. A source file can contain two types of C# code. One kind, which is executed for code generation, and uses the second kind as string inputs to operate on (see Section 4.2.3). This is a feature of the used render system but has no special support in the development system. Debugging this system can still be tricky, but is much easier than C++ templates and is more powerful than C# generics.

This code generation makes it possible to write generic versions of operators, which generate specific functionality in a pre-compile step. For example only one class for vector types has been implemented which generates specific classes for different types (e.g., float and double) and sizes of vectors.

3.6 Summary

This chapter introduced the concepts on which the implementation is based on. The basic iDSL structure is presented, which is the main interface for the shader developer. The shader tree gives an idea about which data structures are needed and the shader fragments lay out the desired feature set of the tree nodes. The overall workflow shows the components involved in the process of getting a hardware shader from a shader tree. The last section further explains some design decisions and pros and cons of alternative ideas. It also introduces ways to extend the current system, which will also be discussed with regard to the concrete implementation in Section 4.8.

CHAPTER 4

Implementation

This chapter describes the implementation of the design. The system uses the programming language C# and as an example target shader language HLSL. To not get distracted by building all the basics of a render engine, an already existing full-featured engine is used. The basic concept of that engine and functionality of HLSL are described in this chapter for the better understanding of the shader framework. Further, implementation details of the framework are described and also methods to further extend them. Additional details of the implemented class hierarchy can be seen in the class diagrams (see Appendix A).

4.1 C# as the Host Language

The main computer language for interactive computer graphics is C++. Most programmers are familiar with it and there are a lot of independent native libraries. Application development shifted towards Java, because it is a multi-platform language, comes with a vast library, and has many modern features like a garbage collector. Java compiles to a byte-code that is interpreted by a virtual machine and is therefore slower than programs written in native machine code.

Graphics programmers shun interpreted languages like Java and C#, because they fear the impact on their application speed. This concern seems outdated. CPU's are becoming increasingly faster and graphics speed is determined through graphics cards and bus systems for data transport. The performance hit of interpreted languages seems especially small when compared with the features modern languages introduce.

4.2 Rendering Engine Aardvark

The VRVis developed a rendering engine which they named Aardvark. It is written in C# with a DirectX® back-end and used for projects in multiple fields, e.g., medical visualization and virtual reality. It has a great amount of features like a dynamic scenegraph, meta programming, an image library, and others.

Aardvark uses file-based shaders, which can be micro or uber shaders. The need for better shader handling and the already existing features made the engine a perfect basis for this thesis.

Figure 4.1: VRVis

Figure 4.2: Aardvark Renderer

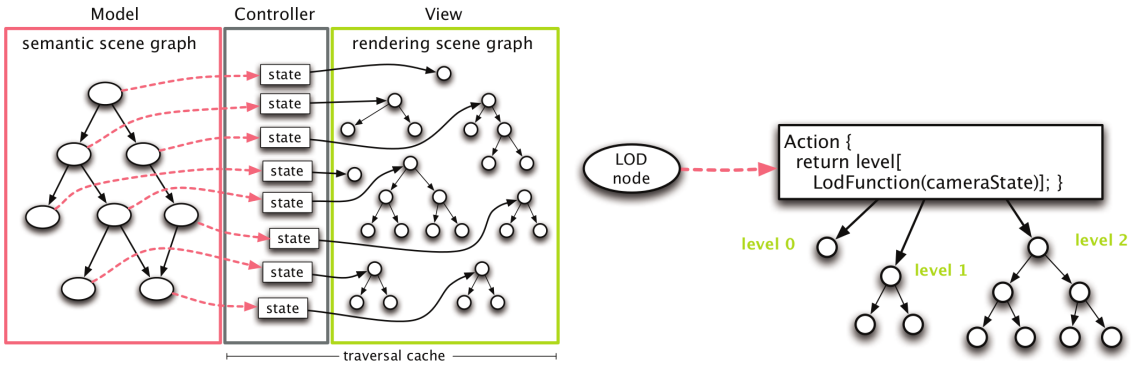
4.2.1 Scenegraph

Aardvarks scenegraph is the main structure of rendering for the system. Nodes of that graph can be 3d-models, the camera, transformations for positioning, and others. The shader trees connect to the scenegraph to get the compiled shader to the graphics card.

The scenegraph is split into a *semantic* and a *rendering graph* [Tob11]. The semantic part is defined by the user and describes the scene. The rendering graph is a forest of small graphs generated from the nodes of the semantic graph and optimized for rendering (see Figure 4.3a).

For each semantic node type exist different *translation rules* to generate their part of the rendering graph. The graph traversal determines which one of the node's translation rules should be used. All of those created rule objects are cached, so every time the same traversal is used, the rule objects update their part of the rendering graph based on the traversal state and their local state.

Example: An old Victorian table could be modeled with four legs and the table-top as the leaves of the semantic scene graph. Each has its material as an attribute. The legs have parent transformation nodes to position them and on top a group node to combine all the legs and the table-top. The legs are level-of-detail nodes and the translation rule creates a rendering graph depending on the distance to the camera saved in the traversal state (see Figure 4.3b).



(a) Concept (b) LOD Node

Figure 4.3: The Aardvark scenegraph [Tob11] facilitates the MVC pattern to separate the user-defined scene graph and the rendering scene graph which gets created on a per-node basis. This is done by translation rules (the controllers), which is shown in the example of the LOD node with an action as such a translation rule.

4.2.2 Hooks

Here is the description of how the shader trees *hook into* the parts of Aardvark's scenegraph (see Figure 4.3a).

Model Each vertex geometry in the *semantic scenegraph* has a surface, which is associated with a specific shader and is used to communicate values from the C# application to the shader. Each complete shader in the iDSL inherits from `ShEffect`, which can be used like any other surface of the system.

View A `SurfaceLeaf` is a node from the *rendering scene graph* representing a surface. The rendering engine already has the functionality to create such a node that can hold, compile, and

upload shader code to the GPU. Therefore this node can be used unchanged with the legacy shader code created from the semantic model.

Controller For each type of node of the *semantic scene graph* there is a `Rule` which knows how to create a corresponding node of the *rendering scene graph*. The `Rule` is called on each traversal of the semantic scene graph, but gets cached, so that it doesn't have to recreate the rendering node. For example the `SurfaceLeaf` is created once and only changed if the shader code would change. To dynamically update shader parameters, the `Rule` checks the default values of the `ShaderEffect` every frame.

The mappings are stored in the `RuleMap`, and there is one `Rule` for each concrete node class. In the shader-tree system, there is only one `Rule` for all shaders, but each shader is a concrete class and therefore has to be registered. This is done by finding all classes inheriting from `ShaderEffect` in the assembly and registering them to the shader `Rule`.

4.2.3 Code Generator

To embed shader development into C#, a lot of similar functionality has to be implemented. For example, the swizzle operator comes in different varieties (`.XYZ`, `.XY`, `.X`, `.Y`,...) for different types (`float`, `int`,...). This can take a lot of time to write out all by hand. Traditionally, the use of generalized programming like C# generics is used to speed up the development and make the code more compact for easier maintenance. Another approach is *meta programming*, where source code can be manipulated by other source code and therefore is able to generate code or do optimizations like loop unrolling.

Aardvark supports such a meta programming facility to generate code. The meta language is defined in comments and evaluated in a preprocessor step as normal C# code. Variables defined in the meta language can be used in the manipulated source code by appending two underscores in front and afterwards the variable name (see example in Listing 14).

Meta programming is used heavily in this thesis for code generation. Four meta classes with about 700 lines of code create 35 classes with about 5.000 lines. This is used to generate the following:

- Classes with different number of fields or template parameters
- Classes with different `where` constraints
- Classes inheriting from classes or interfaces with different template parameters
- Methods with different numbers of parameters
- Loop unrolling

The best example is the shader attribute framework (see Section 4.4.2). Three meta classes are defined for scalar, vector, and matrix attributes. These generate classes for multiple types like bool or float and different sizes of vectors and matrices.

```
//# var types = new[] {
//#     new[] {"ShInt", "int", "Model.ScalarType.Int", "i"},
//#     new[] {"ShFloat", "float", "Model.ScalarType.Float", "f"}};
//# foreach (var t in types) {
//#     var shScalarType = t[0];
// Vectors
//#     for (int i=2; i<5; i++) {
//#         var shType = shScalarType + i;
//#         var aardvarkType = "V" + i + t[3];
public class __shType__ : ShAttribute<__aardvarkType__>
{
    // Swizzle Operator
    //# var type1 = t[0];
    public __type1__ X
    {
        get
        {
            var atom = new ShAtom<__type1__> (
                Model.Atom.AtomType.SwizzleX, this);
            return atom.ReturnValue;
        }
    }
    // Arithmetic Minus
    public static __shType__ operator -(
        __shType__ vec, __shScalarType__ scalar)
    {
        var atom = new ShAtom<__shType__>(
            Model.Atom.AtomType.Subtract, vec, scalar);
        return atom.ReturnValue;
    }
}
//#     } // Vectors
```

Listing 14: Meta-programming example: Comments starting with the character '#' are executed by the meta-programming framework. Variables with double underscores in front and after their name are filled by variables of the same name from the commented meta program. This example would generate six classes (`ShInt2` - `ShInt4`, `ShFloat2` - `ShFloat4`).

4.3 HLSL

To understand some implementation decisions of the shader tree, we have to take a closer look at a shader language. The rendering engine Aardvark uses HLSL as shader language and therefore this was the main source to look at, but the system can be easily adapted to other shader languages.

HLSL is a programming language with basic control structures like if/then/else, while, etc., and user-defined functions [Mice]. Beside typical data types like float and bool, there are also

matrix types like float2x2, texture types, and more. The built-in library offers the needed geometric functions like dot product or texture sampling.

A *preprocessor* like in C and C++ is available. Code separation is possible through the '#include' directive, and different features can be decided at compile time by using '#ifdef'.

Semantics are identifiers attached to variables for communication between stages and the rendering system. From the main application on the CPU the semantics are needed to set global variables and input buffers in the shader. Only a few are predefined for communication with the fixed function parts (e.g., 'SV_Position' for setting the position of the vertex in the vertex shader).

Annotations are additional user-defined informations for variables. They have no impact on the shader, but can be extracted by the host application to gather additional information about the shader. They are written between '<' and '>', and have types like `string` or `int`.

A *DirectX® effect* bundles shaders of different stages, pipeline states, and even multi-pass configurations together. Mostly they are stored in text files with the extension '.fx' (see Listing 15).

```
1  float4x4 MVPT : ModelViewProjTrafo
2                  <string info="i am an annotation";>;
3
4  float4 vertexShader(
5      float4 inPosOS : POSITION, float4 inCol : COLOR,
6      out float4 outCol : vp_0) : SV_Position
7  {
8      outCol = inCol;
9      return mul(inPosOS, MVPT);
10 }
11 float4 pixelShader(float4 inCol : vp_0) : SV_Target0
12 {
13     return inCol;
14 }
15 technique10 Render
16 {
17     pass P0 {
18         SetVertexShader(
19             CompileShader( vs_4_0, vertexShader() ) );
20         SetPixelShader(
21             CompileShader( ps_4_0, pixelShader() ) );
22     }
23 }
```

Listing 15: This simple DirectX® effect example shows a set of pixel and vertex shader.

4.4 iDSL

As described in Section 3.3 the iDSL already describes a shader tree, which is the basis for the semantic model, which is again a shader tree. This chapter describes the basics of the iDSL

shader tree.

The nodes of the shader tree are called fragments (see Section 4.4.1). All fragments have output and input attributes, which hold informations about the connection between fragments (edges of the shader tree, see Section 4.4.2). Each fragment can return its semantic model, which is used for further processing by visitors (see Sections 4.5 and 4.6).

4.4.1 Fragments

`ShFragment` is an abstract class that is the base for a shader-tree node. It defines the basics of a fragment:

- *InstanceName* Can be set by the developer to make it easier to find corresponding elements of the iDSL in the generated legacy code
- *Inputs and Outputs* Lists of `IShAttribute` (see Section 4.4.2)
- `GetShaderModel` Each fragment has to know how to create its corresponding shader model

There are four different types of fragments: groups, atoms, functions, and expressions (see Figure 4.4 or Appendices A.1.1 and A.1.2 for more detailed diagrams).

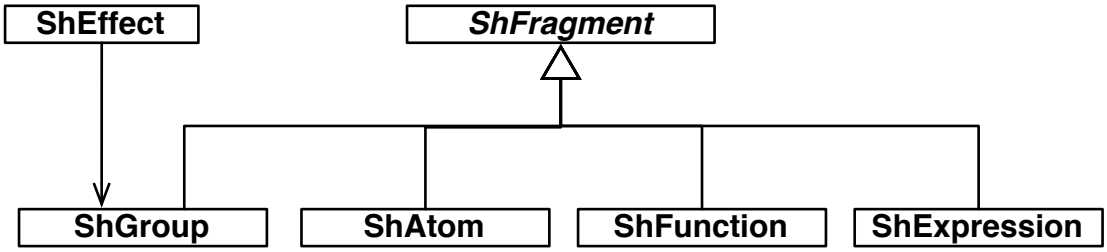


Figure 4.4: The iDSL fragment hierarchy represents all shader tree nodes, with the *ShGroup* having additional functionality to tie in with the *ShEffect* as shader stages

Groups They have a very basic importance to the system. While attributes hold connection informations on creation, groups collect those information on processing. All fragment connectivity has to take part inside a group to get recognized later. Their inputs and outputs regarding the `ShFragment` specification are called external. For each external output exists an internal output. The algorithm defined by fragments inside the group connects its results to those internal outputs (see Figure 4.5).

Atoms They are the smallest fragment entities with only one output attribute, the 'return value'. They are distinguished by an identifier (`AtomType`) which has to be mapped to a concrete functionality by the visitors. They are never created directly by the user. Instead they are generated by functions on an `ShAttribute` like a dot product or swizzling. This binds functionality closer to the variables in an object-oriented way.

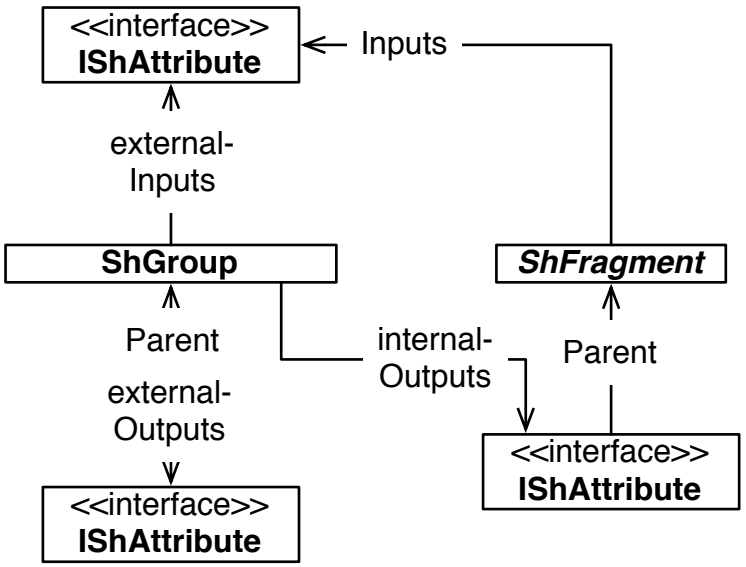


Figure 4.5: The iDSL group structure has three types of *IShAttribute* connections: external inputs, external outputs, and internal outputs. The external inputs are also used as inputs for the *ShFragments* who provide the internal outputs.

Expressions and functions They encapsulate concrete shader code in a string. They have little writing overhead to be quickly created, but are tailored for a specific shading language. This makes them practical for debugging and prototyping but of limited use for production code.

Expressions are meant for short pieces of code that can be defined in one line. Therefore, only one output value is supported in this fragment, which has to be the return value of the given shader piece. Input-/output-parameters are defined with generic type parameters, and the shader expression is given as a string function parameter. Expression classes with different numbers of input type parameters are generated with meta code (see Section 4.2.3).

Functions encapsulate whole shader functions. This is not as convenient as the expressions, but can be practical if algorithms already exist as legacy shader code and should be tested as such. Each function is a class derived from `ShFunction`, has the shader function as a string and has output-/input-parameters defined.

4.4.2 Attributes

Attributes are the connections between shader-tree nodes. A fragment creates an attribute for each of its outputs, which are assigned to the inputs of connecting fragments. They hold the following informations:

- the name of the parent fragment,
- the name they are registered with at their parent fragment, and
- an optional default value.

The base attribute hierarchy is defined by an interface, an abstract class, and a template class (see Figure 4.6 or Appendix A.1.3 for a more detailed diagram). Most concrete attribute classes inherit from the templated class with a C# type as generic type parameter to store a default

value (e.g., `ShFloat3` binds the vector class `V3f` for default values). The abstract base class is needed for attributes that have special default values (`ShArray` and `ShSampler`).

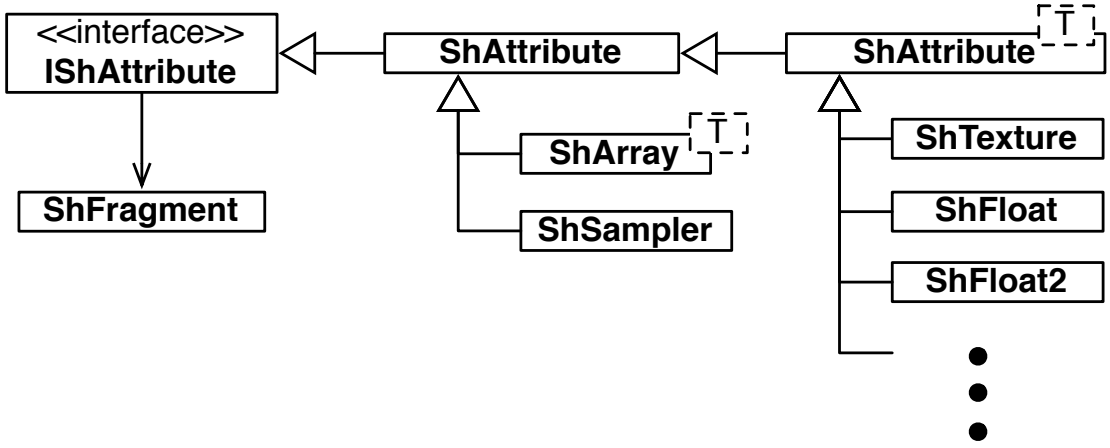


Figure 4.6: This is the iDSL attributes hierarchy. The template *ShAttribute* is used for attributes with a simple default value stored as the template value. The *ShAttribute* is for more complex attributes and the interface for universal access from fragments.

The concrete attribute classes for development are:

- `ShTexture[2D,3D,Cube]`
- `ShSampler`
- `ShArray<IShAttribute T>`
- Auto-generated
  - Scalar `Sh[Bool,Int,Uint,Float,Double]`
  - Vector `Sh[Bool,Int,Uint,Float,Double][2,3,4]`
  - Matrix `Sh[Bool,Int,Uint,Float,Double][2,3,4]x[2,3,4]`

Connections between fragments can be accomplished by assigning an output parameter to an input parameter. The former are created and initialized by their parent fragment and therefore accessed by the destination fragment over the latter.

With the specific types of the system, type errors can be easily found in the host language. Visual Studio even marks such errors in the code, so they can be found while typing. To give the user the convenience of a type system like in a shader language, the system supports swizzling operators and casts.

On shader creation, the default value of an attribute is used to set a default value for the shader variable. During runtime it is checked on every traversal of the semantic scene graph to update the corresponding shader variable.

The default value of attributes based on the template class can be set by assigning a variable or constant of the template type, e.g., a `float` value can initialize a `ShFloat`. This is accomplished by implicit casts.

4.4.3 Constructing a Shader in the iDSL

A complete *shader effect* written in the iDSL to program the GPU consists of several parts. Algorithms are defined with atoms, expressions, and functions, and grouped together by groups. The effects use groups for each shader stage to put a complete effect together. C# control-flow structures like `for` loops or `if` conditions can be freely used, but will be executed to build the semantic model. This loop unrolling is done on the CPU and therefore can only depend on CPU variables. The default value of an iDSL attribute can be accessed at that time, but not their runtime GPU value.

A basic example with part description can be seen in Figure 4.7 and parts of real-life examples can be found in Chapter 5.

Effect Each effect has to inherit from the `ShEffect` base class and has to override the `Link` method. This method defines the linking between the shader stages.

Inputs and outputs of stages to the system as well as global inputs have to be defined and instanced. The effect base class has definitions of the most common ones, but they can be easily extended by inheritance.

After instancing the shader stages and connecting their inputs and outputs, they have to be registered with the system by calling the `Init` method.

```
935 public class TestShader : ShEffect
936 {
937     public new class Globals : ShEffect.Globals
938     {
939         public ShTexture2D Texture = ShTexture3D.Default;
940         public ShSampler Sampler = ShSampler.Default;
941     }
942
943     public Globals Global = new Globals();
944     public VertexInputs VertexInput = new VertexInputs();
945     public VertexOutputs VertexOutput = new VertexOutputs();
946     public PixelOutputs PixelOutput = new PixelOutputs();
947
948     public override void Link()
949     {
950         var vShader = vertexShader.Call(
951             VertexInput.Positions,
952             VertexInput.DiffuseColorCoordinates,
953             Global.ModelViewProjTrafo);
954
955         VertexOutput.Position = vShader.ReturnValue;
956
957         var fShader = pixelShader.Call(
958             vShader.outTexCoord,
959             Global.Texture,
960             Global.Sampler);
961
962         PixelOutput.ImageOutput = fShader.ReturnValue;
963
964         Init(vShader, fShader);
965     }
966     public class vertexShader : ShGroup...
986     public class pixelShader : ShGroup...
1004 }
```

Example for extending globals

Instancing Globals, Inputs, and Outputs of the TestShader

Link inputs and outputs of the shader stages together

Shader stages

Figure 4.7: iDSL shaders are based on *ShEffect*. This example shows extended globals, initialized inputs, embedded shader stages, and their linking.

Group The *shader stages* have to be `ShGroup` fragments (see Listing 16). They inherit from that base class and define their shader code in the `Call` method. Outputs are defined as public member variables and inputs as attributes of the method.

In the `Call` method, the shader stage itself has to be instanced, so the results can be connected to its outputs. In the end, the `InitInputs` method is called to register the inputs and to return the instance of the shader stage. For an example of iDSL shader code see Figure 4.14a.

```
public class TestShaderStage  :  ShGroup
{
    // outputs
    public ShFloat4 ReturnValue;
    ...

    public static TestShaderStage Call (
        // inputs
        ShFloat4 inPosWS,
        ShFloat3 inNormWS,
        ...
        )
    {
        var group = new TestShaderStage();

        // shader code
        ...

        return group.InitInputs(inPosWS, inNormWS,...
            //inputs
            );
    }
}
```

Listing 16: An iDSL shader stage is just another group fragment.

Atoms There shouldn't be a need to create atoms directly. Instead the system makes instances when functions on *attributes* are called, like swizzle operators or geometric functions. These operators can be used like any other C# class member methods.

Expressions Instances of expressions are created by calling the `Call` method with template parameters to define the types of the inputs and the return value. Function parameters are a string defining the legacy shader code and *attributes* for the inputs. The inputs in the string are defined by an underscore and a number corresponding to the position of the *attribute*. The `Call` method returns an *attribute* which represents the return value of the shader string. An example can be seen in Listing 17.

Functions They share some similarities with groups. The base class for functions are `ShFunction` and they contain a `Call` method. The method implementation has only one call to `CreateInstance` to register the inputs and the shader legacy code as string and returns a new instance of this function. The method has the newly defined function class as template parameter. The function is used by calling the `Call` method with the inputs as parameters.

Attributes Variables in the iDSL are instances of `ShAttribute`. All inputs and outputs of fragments are of that type. Assigning C# variables with the corresponding type sets default values for attributes or constant values for fragment inputs. Setting them at runtime triggers an update of the shader attribute in the GPU without the need of recompiling the shader. The Aardvark renderer checks for such updates at each traversal of its scenegraph and takes care of relaying the values to the GPU.

4.5 Semantic Model

The semantic model has a lot of parallels to the internal DSL. It contains all the same information without the overhead that makes the iDSL practical for the developer. The semantic model is focused on presenting the information for further processing (see Figure 4.8 or Appendices A.2.1 to A.2.3 for a more detailed diagram).

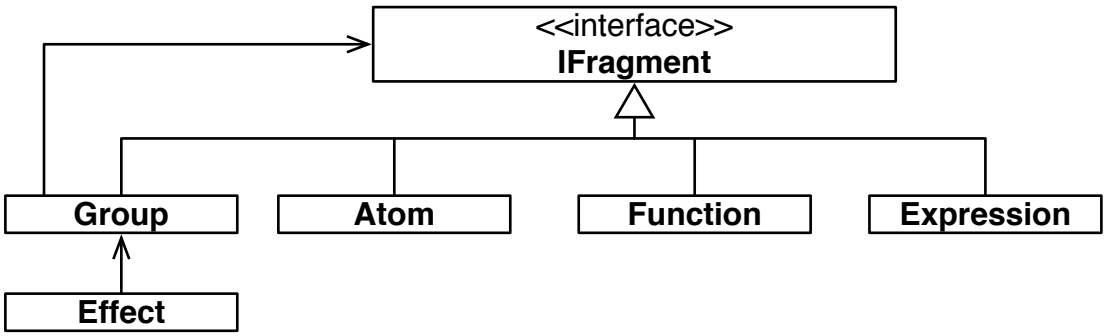


Figure 4.8: The Semantic model class hierarchy has a similar fragment topology as the iDSL, except for the direct use of a fragment interface by the *group*.

All the fragments are represented, but the attributes and type system are highly simplified and therefore auto generated code is not necessary. A simple type hierarchy exists to save type specific data like default values and an `Enum` for a more detailed type description (see Figure 4.9). Instead of the attribute system of the iDSL, the semantic model saves fragment connections in lists of mappings inside the group fragment and the effect.

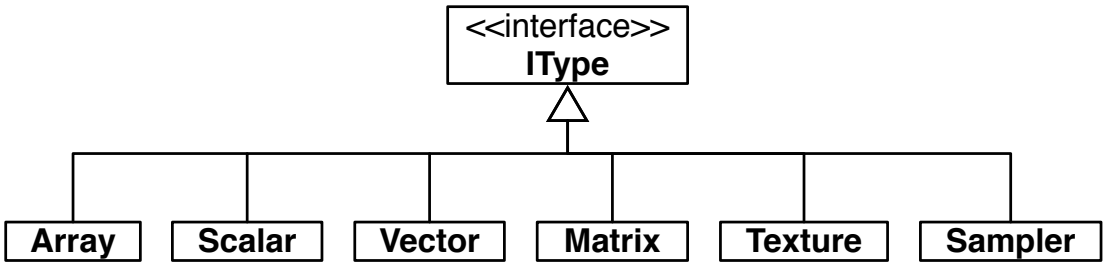


Figure 4.9: Instead of attributes, the semantic model uses a simple *type* structure and safes additional fragment connection information directly in the *groups*.

The model is independent of the iDSL and specific target languages like HLSL, except of the function and expression fragments, which still hold specific code as strings. All fragments and attribute types accept a visitor (see Section 4.6) for further processing.

4.6 Visitors

The `IShModelVisitor` is based on the visitor pattern (see Section 2.5.2) and defines functions to visit all fragments and attribute types of the semantic model (see Figure 4.10). Specific visitors can be created for specific tasks like testing, optimization, or code generation. Visitors might still be independent of any target language and focus on tasks like validation, optimization, and visualization of the semantic model.

At the end of the pipeline is a visitor to execute the semantic model either in a software renderer on the CPU or with a created shader on the GPU. This stage performs optimizations and tests specific for the target language.

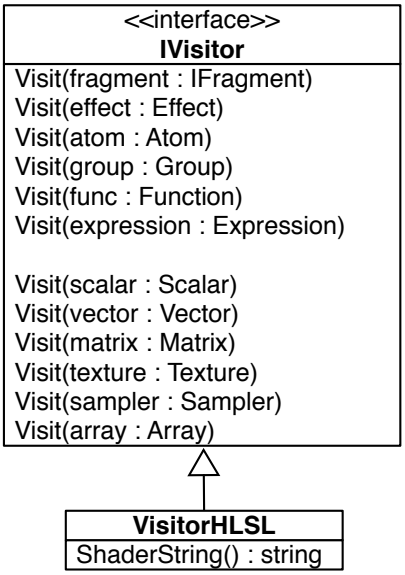


Figure 4.10: Semantic model visitor

HLSL visitor The HLSL visitor creates a DirectX® effect file from the semantic model. The main code generation is done with the `Group`, but there are also some special considerations on `Effect`.

Group All the connection information between the fragments are stored in the group fragment they belong to. The visitor collects this information in a C# `StringBuilder` and creates a shader function with it. The header is based on the group's name and the names of its input and output parameters. The body consists of function calls from the fragments and temporary variables from the output parameters of those calls. `Type` and `Atom` fragments are mapped to shader code with the help of translation tables.

In the iDSL, a fragment instance can be named. All the output variables of that instance get this name as a prefix for better readability of the generated code (see Figure 4.14). If more than one instance has the same name, they get a sequential index as a postfix.

The shader code from an `Expression` and `Atom` fragment is in-lined in the function body created by a `Group`. `Function` and `Group` fragments become functions in the shader code and therefore can be called multiple times. A fragment cache makes sure that they are not generated more than once.

Effect The `Effect` holds one `Group` for each shader stage. The creation of those function headers has to be treated in a special way, because HLSL requires a certain function signature for shader stages to be able to connect the right inputs and outputs. Between two shader stages the output of the first has to coincide with the input of the next in regards to order, type, and semantic naming (see Figure 4.11).

4.7 Development Supporting Features

The iDSL was meant to support the creation of shaders. It does so in several ways. The type system of the iDSL points out type errors before any code is generated. But other errors, especially semantic ones, can only be detected with the generated code.

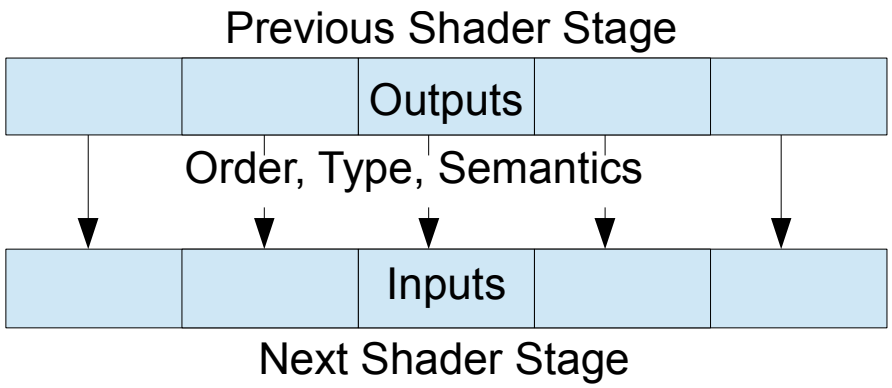


Figure 4.11: HLSL maps inputs and outputs between shader stages by semantics and signature, which is defined by type and order of attributes.

4.7.1 Visual Studio/C# Support

The iDSL is highly dependent on C# to make use of its features and tools. The hierarchical attribute system of this framework, where each abstract shader type is a C# class, makes it possible to get type errors between those attributes in the editor like with legacy C# types (see Figure 4.12).

```
ShFloat3 colorTex = inGrassTex.Sample(inColSampler, inTexCoord * 5);
var blurredTex = Low
var blurredValue = (
var normWS = inNormO

var vecVer2Light = (
var vecVer2Cam = (inCamPos - inPosWS.XYZ).Normalize();
```

(parameter) ShSampler inColSampler

Error:
Cannot implicitly convert type 'Aardvark.Rendering.ShaderTree.ShFloat4' to 'Aardvark.Rendering.ShaderTree.ShFloat3'

Figure 4.12: In-editor live error detection marks the code in question (red wavy line) and shows an error description on mouse over.

```
for (int i = 0; i < pointLightsPos.GetSize(); i++)
{
    var vecVer2Light = (pointLightsPos[i] - position.XYZ).
    var calcLight = CalcLight.Call(normal, vecVer2Cam, vec
                                                          ingEnabled);

    group.dif = (group.dif + (calcLight.dif * pointLightsC
    group.spec = (group.spec + (calcLight.spec * pointLigh
    trans = trans + pointLightsColor[i].W;
}

group.dif = group.dif / trans;
group.spec = group.spec / trans;
```

Normalize
Parent
Reflect
Semantic
SetInstanceName<>
ToString
W
X
XY

ShFloat3 ShFloat3.Normalize()
Creates Atom to normalize vector.

ghtsColor[i].W));
tLightsColor[i].W));

Figure 4.13: In-editor autocompletion pop-up shows further useful code like public methods and fields of an object.

The use of object-oriented programming to bundle functionality into iDSL attributes lets the shader programming use the editor's autocompletion feature while defining shaders. The same feature is also available with fragment inputs and outputs, because they are bundled in the same way (see Figure 4.13).

4.7.2 Readable Generated Shaders

Errors from the shader compiler point to locations in the generated shader code. To make use of those messages, the shader code needs useful formatting and naming. Each atom and group fragment call is put in a single line. This makes it easy to pinpoint errors from the shader compiler based on the line number (see Figure 4.14b).

The developer has the possibility in the iDSL to name instances of fragments. These names are prefixes for the output variables of the fragments in the generated legacy shader code (see Figure 4.14). This makes it easier to understand the generated code and to find the corresponding part in the iDSL.

```
var norm = inNormWS.Normalize()
    .SetInstanceName("normWS");

var vecVer2Cam =
    (inCamPos - inPosWS.XYZ).Normalize()
    .SetInstanceName("vecVer2Cam");

group.ReturnValue =
    new ShFloat4(inCol.XYZ,
        vecVer2Cam.Dot(norm) - 0.3f);
```

(a) iDSL

```
float3 atom = inCol.xyz;
float3 atom0 = inPosWS.xyz;
float3 atom1 = inCamPos - atom0;
float3 vecVer2Cam = normalize(atom1);
float3 normWS = normalize(inNormWS);
float atom2 = dot(vecVer2Cam, normWS);
float atom3 = atom2 - 0.3;
float4 atom4 = float4(atom, atom3);
return atom4;
```

(b) HLSL

Figure 4.14: This comparison of iDSL shader and created HLSL shader has complementary lines marked by the same color. Each fragment of the shader tree is compiled in a single HLSL line, and output attributes are numbered and named if their instance is named in the iDSL.

4.7.3 Live Editing of Generated Shaders

Live editing of generated shaders is a feature of the Aardvark render system and has proven a valuable tool for debugging. The generated shader code is written into a file. If this file changes, it will be reloaded, recompiled and resent to the GPU. This gives the possibility to test fixes right in the generated shader code during runtime of the program.

Those changes are not automatically adopted by the iDSL, but can speed up debugging. The fix has to be mapped back to the iDSL when the error is located in the legacy shader code. The fragment-instance naming (see Section 4.7.2) makes that mapping easier.

4.8 Adding and Expanding

Not all of today's shader features have been implemented. The following sections give directions on how to extend the system.

Most of those features are used during shader development. This needs the right means inside the iDSL and structures to mirror that in the semantic model. Shader features need support in all shader-creation visitors that use them. Other visitors might just ignore the new feature and need no change at all.

The following sections present these feature extensions:

- Additional types

- Additional basic functionality with atoms
- New processing algorithms by adding new visitors
- Extending the iDSL by structs to group variables
- New shader stages (e.g., geometry shader)

4.8.1 Types

The type system consists of the `ShAttribute` hierarchy (see Section 4.4.2 and Appendix A.1.3) on the iDSL side and the `Type` hierarchy (see Appendix A.2.3) on the semantic model side. Most of the types are auto-generated and can be easily extended, while others need more considerations.

Simple types Simple scalar, vector, or matrix types are the easiest extensions. The template which auto-generates those types for the iDSL defines an array for the basic scalar types and generates the scalar type, all vectors, and all matrices for 2 to 4 dimensions (see `ShAttributes_template.cs`).

The semantic model can handle vectors and matrices of any size. Only new scalar types have to be added in the `ScalarType` enumeration (see `ShModelTypes.cs`).

In the HLSL visitor, basic types are mapped to HLSL types with a dictionary (see `ShModelVisitorHLSL.cs`).

Complex types More complex types like textures, array, or sampler are defined as their own classes without auto-generation (see `ShAttributes.cs`). They can have additional information saved, e.g., texture coordinates. In the semantic model, they have to be defined explicitly as well to store that information (see `ShModelTypes.cs`).

Most of the complex types have their own function in the visitor interface (see `IShModelVisitor.cs`). Changes would effect all visitors, although the implementations might not be that complicated. Textures only have one such function based on a common texture class and are further distinguished in the HLSL visitor based on a dictionary.

Structs Structs give the means to group variables together. This is especially useful to group input and output variables together between shader stages. All of today's shader languages support structs and therefore make it easy to implement in the code-generation visitors. In the semantic model, they could be realized as dictionaries. Finally, C# own structs could be used in the iDSL, and with reflection, the names of the variables inside the struct could be extracted.

4.8.2 Atoms

All atoms in the iDSL are based on one atom class, which takes an identifier from the semantic model as constructor attribute (see Appendices A.1.1 and A.2.2). The type of the atom's return value is a template parameter, the input types are constructor attributes.

In the semantic model, a simple enumeration defines all available atom identifiers (see `Model\ShAtom.cs`).

Atoms shouldn't be used directly by the user in the iDSL, but rather be encapsulated in a function. Swizzling or basic algebra like multiplication are defined as atoms created in appropriate functions in the type classes (see `ShAttributes_template.cs`).

The visitor interface has a function for all atoms, and the HLSL visitor uses a dictionary to map the HLSL code to the specific atom.

4.8.3 Visitors

New visitors for e.g., optimizations or target languages can be created by implementing the visitor interface (see Figure 4.10 and `IShModelVisitor`). This interface defines functions to visit a whole effect, all fragment types and different types from the type system. The visitor gets started by calling the effect function.

Traversing the semantic model and all other functionality is all up to the specific visitor. Examples might be:

- An **optimization** visitor might give back an optimized semantic model.
- An **error-checking** visitor might accumulate error messages and their location in the semantic model.
- A **shader-generating** visitor should give back the source code of the defined shader in his target language, e.g., GLSL, Cg.
- A visitor for **execution on the CPU** might execute a software-rendering system during iteration or just generate shaders for that system.

4.8.4 Shader Stages

Shader stages are rather specific and have to be evaluated independently. The current framework has support for vertex and fragment shaders and is missing two stages of today's pipeline: geometry and tessellation.

Geometry stage The geometry stage processes one primitive and might have additional information of the vertices adjacent to it. It generates zero or more new primitives and discards the old one.

Important details of the geometry stage:

- Input primitive (e.g., point, line, or triangle with adjacent vertices),
- Output primitive (point, line, triangle),
- Maximum number of output vertices that will be created,
- Input data per vertex, and
- Output data per vertex (stream).

The geometry stage might be implemented as its own class derived from the `ShGroup`, or additional variables in `ShEffect` store the additional information needed. This information is at least the output primitive type and the maximum number of output vertices.

The input primitive data is stored in arrays. If structs, as described in Section 4.8.1, were implemented, these information can be grouped and put into one array, otherwise all of the input arrays have to be the same size in an array of arrays. The input array size can be mapped to the input primitive type. The output primitive data is defined alongside the input data as attributes of the `ShGroup`.

The output data in the shading languages are streams. Either the visitors track the assignments to guess where one primitive ends and the next should be started, or a new `ShAtom` is introduced to mark it in the iDSL.

4.9 Summary

This chapter gave a deeper understanding of the shader framework. It showed how the system works together with the used render engine and how it uses the engine's advanced features, e.g., meta-programming. While in the previous chapter, the design of the shader tree was laid out, this chapter introduced implementation details and the extensibility of the system.

CHAPTER 5

Example

An example was created to test the development and final implementation of the shader framework (see Figure 5.1). This chapter shows and explains some of the source code to demonstrate the system. The following features were implemented in the example: procedural textures, animated vertex displacement, animated particle system with instancing, and multi texturing. The complete source code can be obtained from the author.

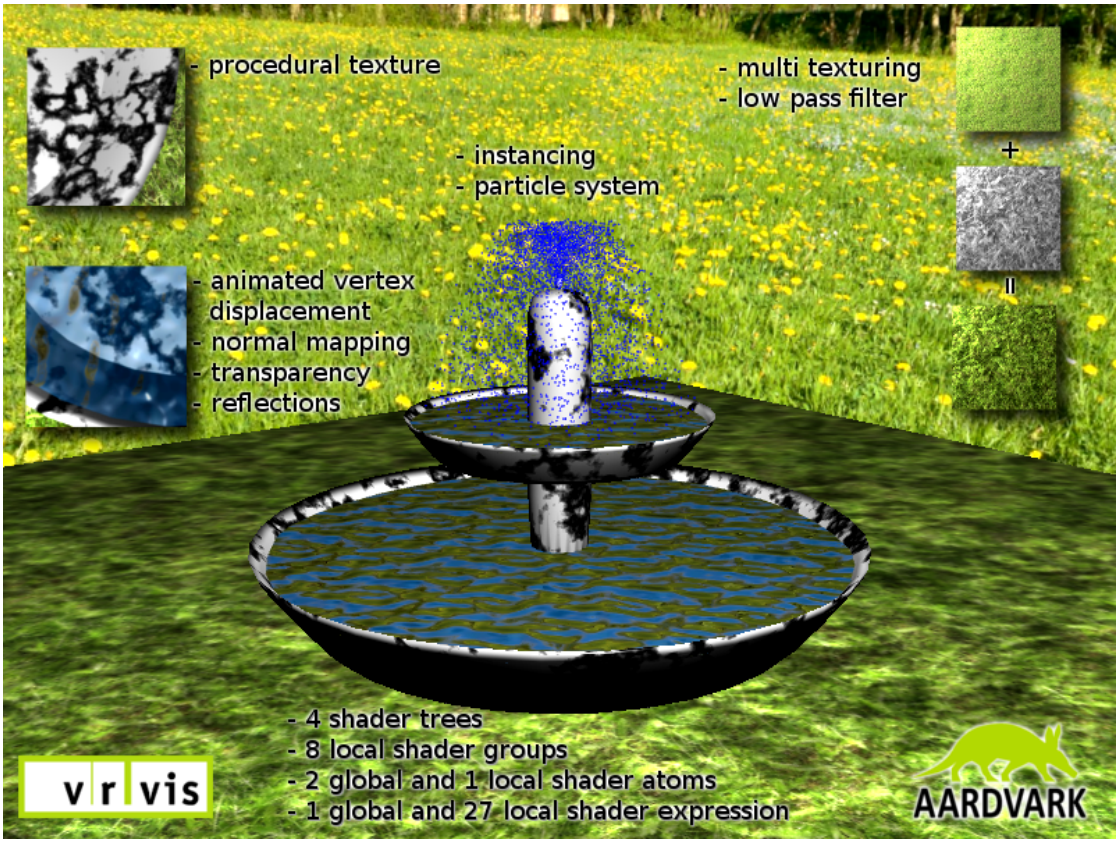


Figure 5.1: The fountain example includes four shader trees for procedural marble, animated water, low pass filtered grass, and an instancing particle system for water droplets.

5.1 Sprinkle Shader

This shader generates drops of water coming out of the top of the fountain.

Instancing is a feature of DirectX® where a shader is run over the same 3d model several times with different transformations to generate different objects that look the same. The simulation of the water drops is done on the CPU, and the resulting transformation is sent to the shader (see Figures 5.2 and 5.3).

```
public class InstancingShader : ShEffect
{
 public new class VertexInputs
  : ShEffect.VertexInputs
 {
  public ShFloat4 ReplicatorTrafoRow0;
  public ShFloat4 ReplicatorTrafoRow1;
  public ShFloat4 ReplicatorTrafoRow2;
  public ShFloat4 ReplicatorTrafoRow3;
 }
 ...
 public class vertexShader : ShGroup
 {
  public ShFloat4 ReturnValue;

  public static vertexShader Call(
   ShFloat4 inPosOS, ShFloat4x4 inMVPTrafo,
   ShFloat4 inTrafoRow1, ShFloat4 inTrafoRow2,
   ShFloat4 inTrafoRow3, ShFloat4 inTrafoRow4)
  {
   var group = new vertexShader();

   var trafoMatrix = new ShFloat4x4(
    inTrafoRow1,inTrafoRow2,
    inTrafoRow3,inTrafoRow4);

   group.ReturnValue =
    inPosOS.Mul(trafoMatrix).Mul(inMVPTrafo);

   return group.InitInputs(inPosOS, inMVPTrafo,
    inTrafoRow1,inTrafoRow2,
    inTrafoRow3,inTrafoRow4);
}}}
```

Figure 5.2: Instancing done in an iDSL shader.

```
//---- Functions ----
float4 vertexShader (
  float4 inPosOS, float4x4 inMVPTrafo,
  float4 inTrafoRow1, float4 inTrafoRow2,
  float4 inTrafoRow3, float4 inTrafoRow4)
{
  float4x4 atom = float4x4(
    inTrafoRow1, inTrafoRow2,
    inTrafoRow3, inTrafoRow4);
  float4 atom0 = mul(inPosOS, atom);
  float4 atom1 = mul(atom0, inMVPTrafo);
  return atom1;
}

//---- Global Parameters ----
float4x4 ModelViewProjTrafo : ModelViewProjTrafo;

//---- Connect stages ----
void InstancingShader_vertexShader (
  float4 Positions:POSITION,
  float4 ReplicatorTrafoRow0:ReplicatorTrafoRow0,
  float4 ReplicatorTrafoRow1:ReplicatorTrafoRow1,
  float4 ReplicatorTrafoRow2:ReplicatorTrafoRow2,
  float4 ReplicatorTrafoRow3:ReplicatorTrafoRow3,
  out float4 Position:SV_Position)
{
  Position = vertexShader(
    Positions, ModelViewProjTrafo,
    ReplicatorTrafoRow0, ReplicatorTrafoRow1,
    ReplicatorTrafoRow2, ReplicatorTrafoRow3);
}}}
```

Figure 5.3: HLSL code created from Figure 5.2.

5.2 Water Shader

The water shader implements several features in the vertex and pixel shader. All of this code is independent of any concrete shader language and demonstrates the power of the iDSL and ShAtom. A normal map with wave patterns is read from a texture, which is shifted on the CPU to create a simple animation (see Figure 5.4).

Vertex displacement In the vertex shader, the vertex position is adjusted by a normal map (see Figure 5.5). The normal map is constantly updated on the CPU to create a continuous motion (see Figure 5.4).

Normal mapping In the pixel shader, the normal map is read, and all the values are shifted so they are positive numbers. The world normal is calculated and adjusted by the values from the normal map (see Figure 5.8).

Reflections The environment uses a cube map for the background. This texture is sampled with a reflection vector based on the adjusted normal. (see Figure 5.8).

Transparency The transparency is simply based on the view angle to the surface (see Figure 5.8).

```
1   var waterShader = new FountainWaterShader();
2   // waveTexs: set of wave textures
3   waterShader.Global.OceanNormalMap = waveTexs[0];
4
5   // animation done with shifting textures
6   Kernel.CQ.EnqueuePeriodic(
7       () => {
8           waterShader.Global.OceanNormalMap =
9               // Kernel.T is time in ms
10              waveTexs[(int)(30 * Kernel.T) % 150];
11      });
```

Figure 5.4: The iDSL variable 'OceanNormalMap' is set periodically to update the HLSL vertex shader global of the same name. The iDSL-group/HLSL-function input variables 'inOceanNormalMap' in the examples Figures 5.5 and 5.7 are set to the updated value and show its use for vertex displacement.

```
var bumpTexTS = inOceanNormalMap.SampleLevel
    (inColSampler, inPosOS.XY / 50, 0).Z * 25;

group.ReturnValue =
    new ShFloat4(
        inPosOS.X,
        inPosOS.Y,
        inPosOS.Z + bumpTexTS,
        inPosOS.W)
    .Mul(inMVPTrafo);
```

Figure 5.5: Vertex displacement done in the vertex shader.

```
float bumpTexTS = OceanNormalMap.SampleLevel
    (ColorSampler, inPosOS.xy / 50, 0).z * 25;

return mul(
    float4(
        inPosOS.x,
        inPosOS.y,
        inPosOS.z + bumpTexTS,
        inPosOS.a),
    ModelViewProjTrafo);
```

Figure 5.6: Hand written HLSL, which is equivalent to Figure 5.5.

```
float4 vertexShader (
    float4 inPosOS,
    Texture2D inOceanNormalMap,
    SamplerState inColSampler,
    float4x4 inMVPTrafo)
{
    float atom = inPosOS.x;
    float atom0 = inPosOS.y;
    float atom1 = inPosOS.z;
    float2 atom2 = inPosOS.xy;
    float2 atom3 = atom2 / 50;
    float4 atom4 = inOceanNormalMap
        .SampleLevel(inColSampler, atom3, 0);
    float atom5 = atom4.z;
    float atom6 = atom5 * 25;
    float atom7 = atom1 + atom6;
    float atom8 = inPosOS.a;
    float4 atom9 = float4(atom, atom0, atom7, atom8);
    float4 atom10 = mul(atom9, inMVPTrafo);
    return atom10;
}
```

Figure 5.7: HLSL code created from Figure 5.5.

```
var bumpTexTS = inOceanNormalMap
    .Sample(inColSampler, inPosOS.XY / 50);
var normWS = new ShFloat3
    (bumpTexTS.X, bumpTexTS.Y, inNormOS.Z)
    .Mul((ShFloat3x3)inMTrafoTI).Normalize();

var vecVer2Cam = (inCamPos - inPosWS.XYZ)
    .Normalize();
var reflVec = (-vecVer2Cam).Reflect(normWS);
var reflectionTexTS =
    inEnvMap.Sample(inColSampler, reflVec).XYZ;

var transparency = 1.5f -
    normWS.Dot(vecVer2Cam).Abs();
group.ReturnValue = new ShFloat4
    (reflectionTexTS, transparency);
```

Figure 5.8: Pixel water shader is calculating reflection and transparency.

```
float4 bumpTexTS = OceanNormalMap
    .Sample(ColorSampler, inPosOS.xy / 50);
float3 normWS = normalize(mul(
    float3(bumpTexTS.xy, inNormOS.z),
    (float3x3)ModelTrafoTransposedInv));

float3 vecVer2Cam = normalize(
    CameraLocation - inPosWS.xyz);
float3 reflVec = reflect(-vecVer2Cam, normWS);
float3 reflectionTexTS =
    EnvMap.Sample(ColorSampler, reflVec).xyz;

float transparency = 1.5 -
    abs(dot(normWS, vecVer2Cam));
return float4(reflectionTexTS, transparency);
```

Figure 5.9: Hand written HLS, which is equivalent to Figure 5.8.

```
float4 pixelShader (float4 inPosOS, float4 inPosWS,
    float3 inNormOS, float3 inCamPos,
    Texture2D inOceanNormalMap, TextureCube inEnvMap,
    SamplerState inColSampler, float4x4 inMTrafoTI)
{
    float3 atom = inPosWS.xyz;
    float3 atom0 = inCamPos - atom;
    float3 atom1 = normalize(atom0);
    float3 atom2 = -atom1;
    float2 atom3 = inPosOS.xy;
    float2 atom4 = atom3 / 50;
    float4 atom5 = inOceanNormalMap
        .Sample(inColSampler, atom4);
    float atom6 = atom5.x;
    float atom7 = atom5.y;
    float atom8 = inNormOS.z;
    float3 atom9 = float3(atom6, atom7, atom8);
    float3x3 atom10 = (float3x3)inMTrafoTI;
    float3 atom11 = mul(atom9, atom10);
    float3 atom12 = normalize(atom11);
    float3 atom13 = reflect(atom2, atom12);
    float4 atom14 = inEnvMap
        .Sample(inColSampler, atom13);
    float3 atom15 = atom14.xyz;
    float atom16 = dot(atom12, atom1);
    float atom17 = abs(atom16);
    float atom18 = 1.5 - atom17;
    float4 atom19 = float4(atom15, atom18);
    return atom19;
}
```

Figure 5.10: HLSL code created from Figure 5.8.

5.3 Marble and Grass Shader

The last two shader examples have their main functionality in a `ShFunction` and a `ShExpression`. They demonstrate how existing shader code can be easily integrated. The concrete `ShFunction` only contains the shader code and the input and output definitions.

Marble shader The marble shader generates a *procedural texture* with Perlin noise based on a random texture. This is all done in the pixel shader, and the vertex shader only forwards inputs and does the basic position transformations.

It uses the `ShFunction NoiseFromStatic3D` to sample the random texture and finally a `ShExpression` to get the marble look (see Listing 17). This can be sent directly to the frame buffer or refined with further lighting calculations.

```
1 var noise = NoiseFromStatic3D.Call(
2     inRandomTex, inColSampler,
3     inPosOS.XYZ, inPerlinNoiseSamples);
4
5 var randomTex = ShExpression<ShFloat>.Call(
6     "1. - clamp(cos(_0.x * 16 + _1.x * .1), 0, 1)"
7     , noise.ReturnValue, inPosOS);
```

Listing 17: Perlin noise is generated from a random texture.

```
1  float4 pixelShader (float4 inPosOS, Texture3D inRandomTex,
2      SamplerState inColSampler, int inPerlinNoiseSamples)
3  {
4    float3 atom = inPosOS.xyz;
5    float NoiseFromStatic3Dcall = NoiseFromStatic3D(
6      inRandomTex, inColSampler, atom, inPerlinNoiseSamples);
7    float expr = 1. - clamp(
8      cos(NoiseFromStatic3Dcall.x * 16 + inPosOS.x * .1), 0, 1);
9    float4 atom0 = float4(expr, expr, expr, 1);
10   return atom0;
11 }
```

Listing 18: HLSL code created from Listing 17.

Grass shader The grass texture is sampled with low pass filters in the `ShFunction LowPassFilter` and then used as a black and white layer added to the original texture in a different resolution, i.e. the filtered b/w texture covers more space than the original texture (see Listing 19). This *multi texturing* hides artifacts created by patterns.

```
var blurredTex =
    LowPassFilter.Call(inGrassTex, inColSampler, inTexCoord);
var blurredValue =
    ( blurredTex.ReturnValue.X
    + blurredTex.ReturnValue.Y
    + blurredTex.ReturnValue.Z) / 3;

var colorTex = inGrassTex.Sample(inColSampler, inTexCoord * 5);

group.ReturnValue = colorTex * blurredValue;
```

Listing 19: Multitexturing done with a fragment for low-pass filtering.

```
float4 pixelShader (float2 inTexCoord, Texture2D inGrassTex,
    SamplerState inColSampler)
{
  float2 atom = inTexCoord * 5;
  float4 atom0 = inGrassTex.Sample(inColSampler, atom);
  float4 LowPassFiltercall =
    LowPassFilter(inGrassTex, inColSampler, inTexCoord);
  float atom1 = LowPassFiltercall.x;
  float atom2 = LowPassFiltercall.y;
  float atom3 = atom1 + atom2;
  float atom4 = LowPassFiltercall.z;
  float atom5 = atom3 + atom4;
  float atom6 = atom5 / 3;
  float4 atom7 = atom0 * atom6;
  return atom7;
}
```

Listing 20: HLSL code created from Listing 19.

5.4 Overview

The following table shows an overview of the used shader-tree fragments in the examples. At least two groups are needed at least for the vertex and fragment shader. This is an implementation decision and the design could also allow other types of fragments for each shader stage.

The examples had four fragments for re-usability: one group and three functions. They were used for basic phong shading, a point light, a low-pass filter, and to generate Perlin noise from a random texture.

Shader	Group	Atom	Func	Expr
Sprinkle	2	15	0	0
Water	3	83	2	0
Marble	2	13	2	2
Grass	2	21	2	1

5.5 Summary

This chapter showed the use and strengths of the system. Besides already integrated features in the iDSL like shader types with constructors and basic arithmetic functionality like multiplication, also other features were integrated, e.g., the function `clamp`. Although the system should be further extended (see Section 4.8), many features can already be used in rapid prototyping before they are finally included in the iDSL.

CHAPTER 6

Evaluation

To evaluate the created system, it is compared to legacy shader code and other presented shader frameworks (see Chapter 2). Different categories are investigated, like compile time, code complexity, and different feature sets.

6.1 Comparison to Legacy Shader Code

To evaluate the presented framework, the effects from the examples were also implemented directly in HLSL. A comparison of code complexity and compile time was done. Also some thoughts are given on the matter of run-time and debugging.

6.1.1 Complexity

The following table compares the shader examples written in the iDSL with the manually optimized HLSL code. The first comparison takes a look at the lines of code that define the algorithm, and the second one compares the necessary overhead like class and function definitions. Comments and empty lines were removed, and the same formatting style was used for iDSL and HLSL code. A visual representation of that dataset can be seen in Figure 6.1, with the overhead count on top of the lines of code and the corresponding iDSL and manual implementations next to each other.

	lines of code			lines overhead		
Shader	iDSL	manual	factor	iDSL	manual	factor
Marble	28	29	0,97	67	46	1,46
Grass	42	43	0,98	69	46	1,50
Water	29	39	0,74	68	47	1,45
Sprinkle	9	9	1,00	43	38	1,13

The comparison shows that algorithms can be defined as compact in the iDSL as in legacy shader code and involve less than one and a half as much overhead.

The smaller the fragment, the more overhead it has compared to lines of algorithmic code. The smallest function fragment that was used in the example (not explicitly shown in the table) has four times the overhead in the iDSL compared to the HLSL code. These smaller fragments

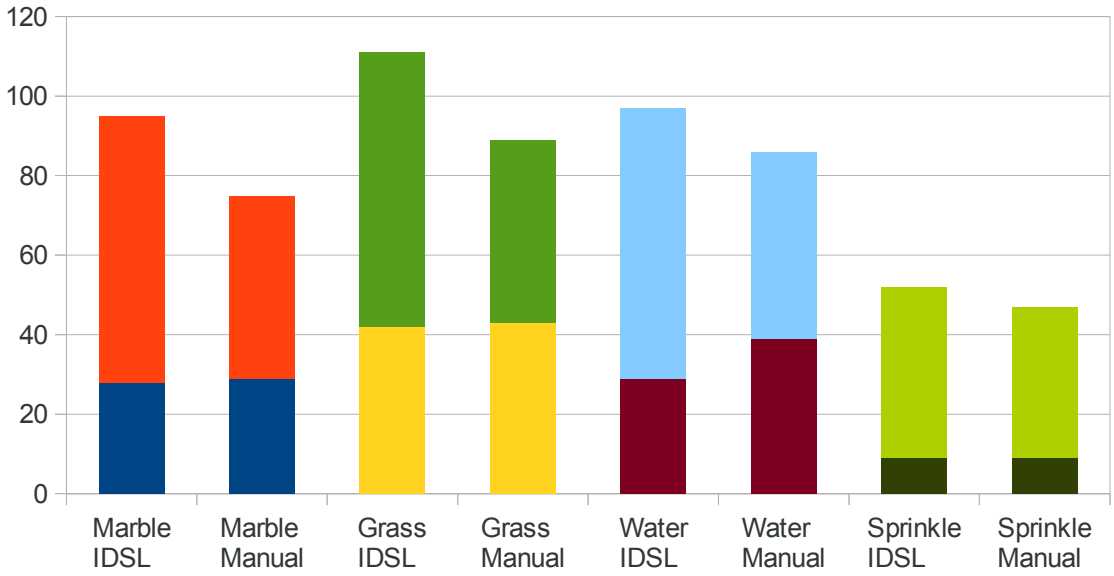


Figure 6.1: Lines of code (lower half) and Overhead (upper half) comparisons of iDSL shader and manually written HLSL shader is shown here. The shaders with the same functionality share the same color scheme.

are meant for heavy reuse, therefore the overhead has lesser significance, as seen in the comparison table, where the mentioned function fragment is used in all of the examples, except the sprinkle shader and none of the overhead factors is higher than one and a half.

6.1.2 Compile Time

Measurements were performed on an Intel Core i7 3,4 GHz with 16 GB main memory. Times were taken for the translation of the iDSL to the semantic model (SM), from the model to HLSL shader code, and finally the HLSL shader compiler (GPU). A visual presentation can be seen in Figure 6.2, where compile times are compared as percentage of the whole process of compiling a shader from an iDSL. The table gives the factor of overall time compared to the time necessary without iDSL and semantic model.

Shader	SM	HLSL	GPU	factor
Sprinkle	2ms	1ms	11ms	1.22
Water	11ms	5ms	40ms	1.40
Marble	8ms	1ms	28ms	1,33
Grass	7ms	1ms	85ms	1,09

Although a much more convenient programming model is provided, the overall compile time impact is moderate: The additional overhead introduced by the framework to generate HLSL code is relatively low (a fraction of a second) even for complex shaders, making the approach feasible for interactive editing of material parameters. Note that recompiling shaders each frame is not applicable anyway, since the compilation time for HLSL effects (from HLSL to GPU byte code) lies around 10-90ms for small programs as well.

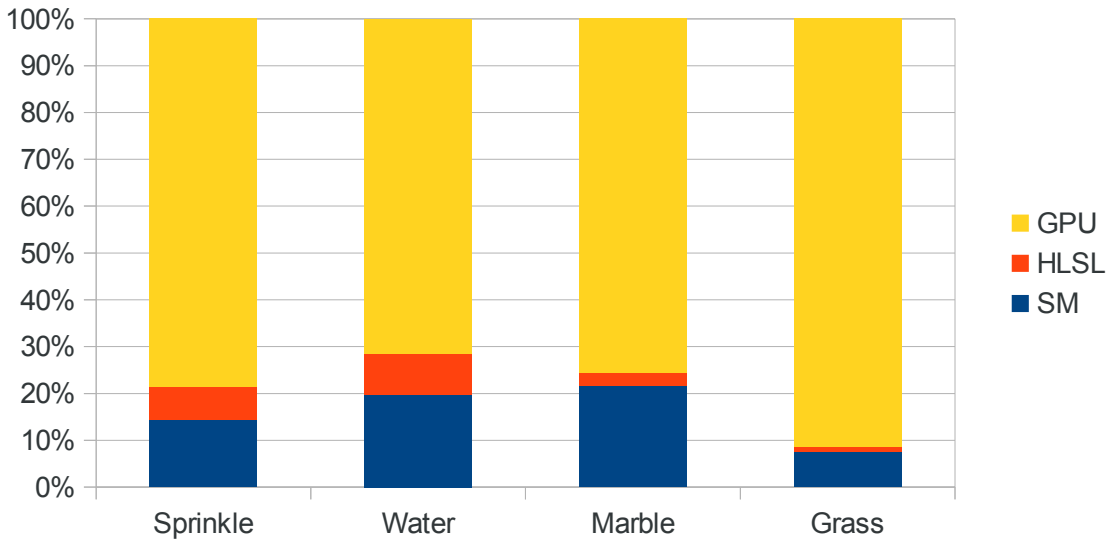


Figure 6.2: This graph compares compile stages (iDSL to shader model, to HLSL, and to GPU shader code) as fractions of the compile time for the whole process.

6.1.3 Run Time

The rendering time needed for a generated shader was compared to a manually optimized one. On the above described computer, the test scene took about 7.4 ms/frame with animation and 3.8 ms/frame without animation to render with fluctuations of about a quarter ms. Between the generated and manually optimized shader no differences could be observed in this setup. The presented examples where used, but neither they nor the used scene was meant for performance testing. Therefore, testing of bigger scenes with more shader effects would be necessary for a representative performance study.

6.1.4 Debugging

The proposed framework introduces additional possibilities for debugging shaders:

Host language The development environment of C# in Visual Studio supports finding errors in the iDSL already while writing code, e.g., by highlighting type errors or typos (see Section 4.7.1). Basic error checking of the shader-tree structure is done during conversion to the semantic model, but can be extended with any type of run-time checking with new visitors. This can even be used to run the shader code on the CPU in a software renderer to use CPU tools for debugging (see Section 4.8.3).

Shader language The generated code is legacy shader code and therefore it can be debugged like any other HLSL code. The readability of the generated code is key to understand the location of the problem and is the responsibility of the visitor. Another key feature to trace an error from the shader code back to the iDSL is the ability to name a fragment call, which is used in the generated HLSL code to name the output variables of that fragment accordingly. This supports matching parts of shader code to the iDSL it is based on (see Section 4.7.2).

6.2 Comparison to Other Solutions

Further evaluation is done by comparing the proposed solution with other works in specific categories. The following frameworks have been mentioned in Chapter 2, but will be described further here:

LibSh The Sh library embeds shader development into C++ [MQP02, MDP+04]. A more detailed description is given in Section 2.3.3.

Vertigo Vertigo is embedded into the functional language Haskell [Ell04]. A short description is given in Section 2.3.3.

Renaissance Renaissance is a functional shading language [AR05].

Spark Spark implements an object-oriented shading language [FH11].

p.s. Is the proposed system of this thesis.

Following table shows the solutions compared in certain categories. A closer explanation and further defaults of the categories follow there after.

	p.s.	Spark	Renaissance	Vertigo	LibSh
Type of composition	Procedural	OO	Functional	Functional	Algebra
Composition granularity	Stage	Pipeline	Pipeline	Stage	Stage
Metaprogramming	yes	no	no	yes	yes
Type system[1]	t.i. partial	e. static	t.i. static	t.i. static	e. static
Low-Level optimization	yes	no	no	no	no
Target	modular	HLSL	GLSL	gpu	gpu/cpu
Pluggable Optimization	yes	no	no	no	no

Type of composition This category mainly describes the programming paradigm. Embedded languages usually inherit this from the host language. Although LibSh is embedded in the object-oriented C++, it is implemented for procedural composition. It was later extended with further operators to combine or connect shader pieces. This is called a shader algebra.

The proposed system is embedded in C#, which is a multi-paradigm language, but is implemented with a procedural paradigm in mind and using object-oriented structures for grouping and simplifying the handling of code modules. This approach gives the shader developer a familiar feeling to programming procedural hardware shaders.

Composition granularity Hardware shaders split the shading pipeline into stages (see Section 2.2.1). A pipeline approach is done by Spark and Renaissance with the use of computational frequencies (see Section 2.4.1). This is accomplished by binding variables instead of whole code pieces to certain stages, and the framework splits the rest of the code in the end. Therefore, shader algorithms fluently span the whole pipeline.

The proposed system creates shaders by stage. This approach was chosen for its simplicity, but a future extension to a pipeline approach was held in mind. This would need an extension

[1] t.i. = type inference, e. = explicit

of the attribute system to allow tagging for computational frequencies, and the visitor for code generation has to be extended for stage splitting.

Metaprogramming Metaprogramming is the ability of a program to write or manipulate other programs. Functional languages have a long history of metaprogramming, and C++ can use templates for this purpose. Most iDSLs naturally support metaprogramming.

In the proposed system, all C# commands are allowed alongside shader definitions, but the programmer must be aware that those constructs are evaluated when the shader is created. This brings features like loop unrolling without further implementation efforts.

Type system Most systems have an explicit static type system. Renaissance and Vertigo use type inference for developer convenience.

The proposed system can use C#'s static type system and its type inference. The only exception are prototype nodes. They introduce legacy shader code, which are not type checked in C#, but in the shader compiler.

Low-level optimization All presented systems abstract shader development to be independent of a specific target language. Higher-level optimizations are done automatically, and optimizations for the targeted hardware are left to the shader compiler.

The proposed system adds the prototyping nodes to make additional optimizations possible, which can target specific problems. The use of those nodes is optional, and therefore the programmer can decide if language independence or specific optimizations are prioritized for specific projects.

Target The typical final output of a modern shader system is a program that can be sent to the graphics card for rendering. Older frameworks (LibSh and Vertigo) had an assembly language as hardware shader target, while modern ones create HLSL, GLSL, or Cg code. A special benefit of such shader frameworks is the support of different targets or to run as part of a CPU render pipeline (LibSh).

The proposed system uses the semantic model as an intermediate definition of a shader, so it is independent of specific targets (with the exception of prototyping nodes), and the visitor pattern supports modularized extensions.

Pluggable optimization A benefit of most shader frameworks is additional information of their abstract shader definition that can be used for optimizations beyond the possibilities of a legacy shader compiler. Examples for such information are computational frequencies in Spark and Renaissance, or capabilities from the framework like functional optimization techniques used in Vertigo and Renaissance. Those features are typically high level to be independent of a specific target and hard wired.

The proposed system can use the visitor pattern to implement high-level optimizations, but can also have target-specific optimizers, which can be used when appropriate.

6.3 Summary

This chapter showed a comparison of the implemented system to legacy shader code and existing shader frameworks. It highlighted the benefits as well as some shortcomings for future considerations.

CHAPTER 7

Future Work

This chapter describes further features and ideas from the original research and how they fit into the proposed shader framework. All the components of the system were designed and implemented with future extensions in mind.

7.1 Type System

The type system is one of the key features of this system. It already binds together the whole shader tree and gives basic type checks in the iDSL, but it has even more potential.

The type system shouldn't copy from any shader language but model shader types in an abstract way. A tag system can add hints to the generic types for different optimizations, e.g., for space- or frequency-awareness.

Space-aware tags specify the coordinate space a variable belongs to. Besides making an algorithm easier to understand by adding this additional information, there are also some additional features that can be implemented. A function can check if all its input variables are in the same space or in the right space if it requires a specific one for its calculations. The system could even do automatic space conversions to fix such errors (similar to [MSPK06, JFLC07]).

A frequency-aware type system is very powerful and was already introduced in Section 2.4.1. In this system attributes could be tagged for the frequency needed for a calculation. This makes explicit shader stages obsolete. The code generating visitors have to do the splitting based on the tags and the available stages of the target shader language. This makes more automatized optimizations possible.

7.2 Back-end

This includes features that are located after the semantic model in the workflow, such as enhancing existing or adding new visitors. There is still a lot of room for optimizations and extensions:

- The field of computer languages has a vast knowledge of optimizations for language trees that could be performed on a shader tree.

- Further shader language targets can be implemented like GLSL to extend the platform reach of the system.
- A software renderer as target makes further detailed debugging possible e.g., stepping through the shader code with the C# debugger.
- The generated code can be made more readable by a code generator that *in-lines* certain code. In the current system every fragment get its own line of code. The other extreme in-lines too much code and generates long lines of nested code. Small fragments like atoms and expressions can mostly be in-lined. Exceptions are named instances for debugging, that should generate named output variables. A compromise hast to be found to make the code more readable.
- So far the generated code can be read and changed on the fly for debugging. To take things even further an editor could show the iDSL and the rendered shader, which reacts live on changes to the iDSL. Visualization of the iDSL could be done with a graphical editor of the shader tree (see [MSPK06, AW90]) or an editor showing parts of the C# code. The first solution is more robust, but the second one makes applying those changes to the original source code much easier.

7.3 Front-end

This section describes enhancements of the iDSL. Features that take effect in the generated shader code also require some changes in the semantic model and also some visitors.

Branching The embedding of shaders in C# brings all its control structures like branching and loops to the iDSL, but those are all evaluated during shader creation. A lot of algorithms need such constructs in the shader. The front-end should be expanded to support the definition of loops and branching in the iDSL, which are not evaluated on creation of the semantic model but on runtime in the shader. A level-of-detail algorithm might even use both types of branching: dynamic and static to choose types of algorithms on creation and refinement of those algorithms on runtime.

Automatic encapsulation of legacy code `ShFunctions` make it easy to insert already existing code for rapid prototyping. Currently those functions have to be presented as a string and the developer has to define inputs and outputs. This could be automated by a parser which analyses the header of the legacy function and generates an iDSL class encapsulating this code automatically.

Auto forward Another useful simplification of the iDSL is automatically forwarding of variables. Often an input is only used in a later shader stage and patching such attributes through previous stages can be automated. This can be taken further with the addition of default shader stages, e.g., a vertex shader that only forwards inputs and does basic position transformations.

7.4 Shader Features

Besides the already mentioned basic constructs like branching, there are also higher-level shader features that should be considered in future implementations.

Multi-pass rendering is a feature that most complex render engines use, e.g., in games. On iDSL level this could be as easy as concatenating shaders and the back-end handles the details. One of those details is different *render targets*. Currently there has to be always a fragment shader that writes into the frame buffer. Render to texture is needed for those advanced methods.

Further shader stages Another important extensions are further *shader stages* like geometry shader, tessellation, and probably more to come on future GPUs. The basic implementation steps are lined out in Section 4.8.4.

7.5 Summary

While the example (see Chapter 5) demonstrated the power of the already implemented features, this chapter showed the future potential and possibilities for further expanding the system. Each component of the workflow was considered by the design to be extended.

CHAPTER 8

Conclusion

Instead of focusing on a specific feature, this thesis introduced an extensible framework as a base for multiple features, e.g., C# integration, hardware-shader independence, or Visual Studio support. This chapter gives an overview of the system's features and evaluates how challenges (see Section 1.3) were met.

8.1 Features Overview

This section lists the features of the implemented system. The list is divided into following categories: the *iDSL* as the programmers front-end of the system, *C# integration* lists the benefits of choosing C# as the host language of the iDSL, and the *Back-end* covering the semantic model and the visitor system,

iDSL

- It is hardware-shader-language independent, but for quick prototyping hardware shader language code can be injected (see Section 3.2.5 and for an example see Section 5.3).
- The iDSL type system builds upon C#'s strong static type system (see Section 4.7.1) to accomplish following features in Visual Studio:
  - Warnings of incompatible types (without explicit compiling)
  - Autocompletion with code suggestions while typing

C# integration.

- OOP is supported, e.g., classes are used for grouping code and code reuse is done by class inheritance (see Section 3.2.3).
- Metaprogramming can be done by using C# control structures (e.g., looping and branching) to generate different shaders during runtime based on one iDSL shader (see Section 3.2.6).

Back-end

- Different iDSLs can be supported (see Sections 3.1 and 3.5.1).
- Extensibility is accomplished through a plug-in system (see Section 4.6), e.g., to generate different hardware shaders, optimizations, or CPU emulation.
- The system generates readable code (see Section 4.7.2).

8.2 How Challenges Were Met

In Section 1.3, the challenges were presented and an evaluation of the system regarding those challenges follows.

Developers view The iDSL was designed to mimic an actual shader language with support of shader-variable operations like matrix multiplication or swizzling (see Listing 5).

But some new features can feel alien, like grouping through classes (see Section 3.2.3) or fall short on imitation, like calling a *group* class like a function by executing a class method (see Listing 6).

The embedded shader code builds the shader tree and doesn't directly compute those shader statements. This might not be obvious, because it was designed to feel like legacy code. For example a C# loop can be used in the shader abstraction, but gets unrolled on the CPU during the creation of the shader tree (see Section 3.2.6). To get a real loop on the GPU, a shader-tree node for that purpose has to be introduced (see Section 7.3).

Enhanced toolchain Facilitating features from an already existing toolchain is one of the biggest benefits from this system. C# has a great toolchain and it is of great value to make it available for shader development (see Section 4.7.1).

Autocompletion with code suggestions are really helpful with shaders, whose language specifications are constantly growing.

The framework's type system was also of great value with warnings of incompatible types while programming. This made it easy to spot errors without explicitly compiling the whole code and made development faster.

Hardware-shader abstraction Although only a HLSL creation visitor was implemented, the system was designed for the ability to generate code for different types of shader languages (see Section 4.6). The render system Aardvark was a great base system, but fell short in the support for different hardware shaders. Therefore, no other creation visitors were implemented.

The support for injecting hardware-specific shader code into the iDSL was a great feature for early testing and should be viable for prototyping (see Section 3.2.5). It can become a problem if a developer uses such fragments in other group fragments. Hidden that way, it might be forgotten until someone tries to create shaders of another target language. This might lead to hard-to-find errors. A safeguard could be to label the hardware-specific fragments with the type of shader language it holds, so that it can give an explicit error if it is used for an incompatible language.

Abstract shader representation The multi-layered approach with an iDSL, semantic model, and visitors proved to be an excellent technique for a shader framework (see Section 3.1). The iDSL can wholly focus on user input and language integration. The shader tree of the semantic model is concentrated on the abstraction of a shader definition, and each of the visitors only has to be concerned with its specific task.

Such a framework should be a cooperation of software engineers, computer language experts, and shader developers. The semantic model falls into the field of computer languages and would benefit greatly from the knowledge of that field, especially with general optimization visitors. Shader generation and specific optimizations need a shader developer to achieve the best output. The software engineer is best suited for the integration of the iDSL in the host language, because he is trained in designing APIs and bigger systems. This thesis tried to gather the knowledge of those fields, but there is still potential for improvement (see Chapter 7).

8.3 Final Summary

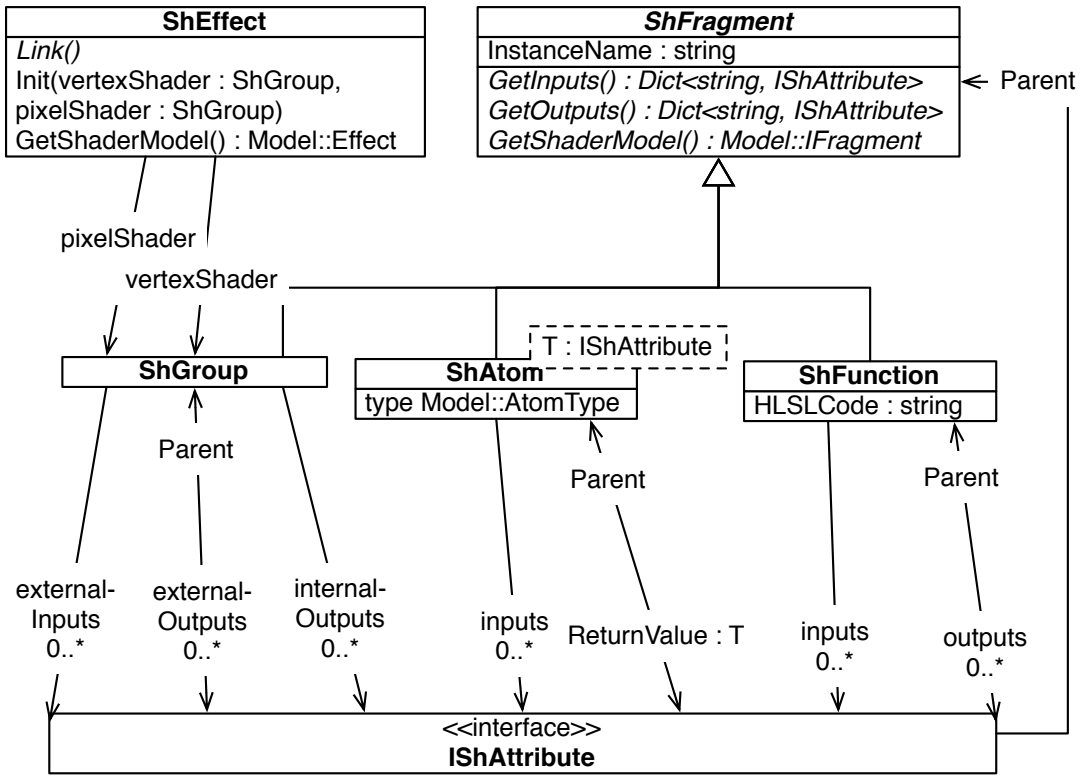
This thesis presented the design and implementation of a shader infrastructure and abstraction layer. The design facilitates common design patterns to build a strong infrastructure. With the use of an *internal domain-specific language*, shader development is abstracted to be independent of a legacy shader language, while offering higher-level features of the host language C#. Although not all desired features could be implemented in the time given, a good framework for future enhancements was created.

APPENDIX A

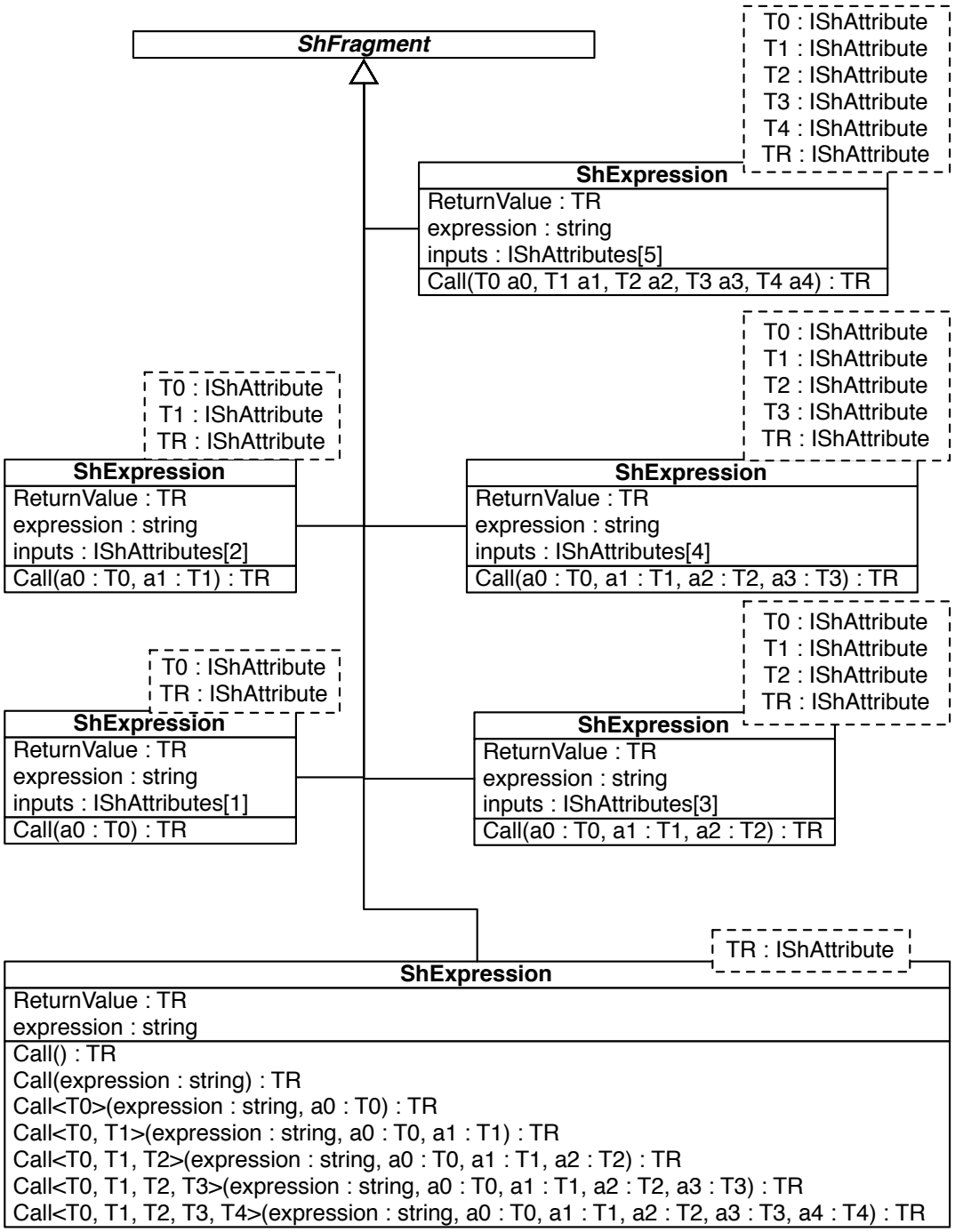
Class Diagrams

A.1 iDSL

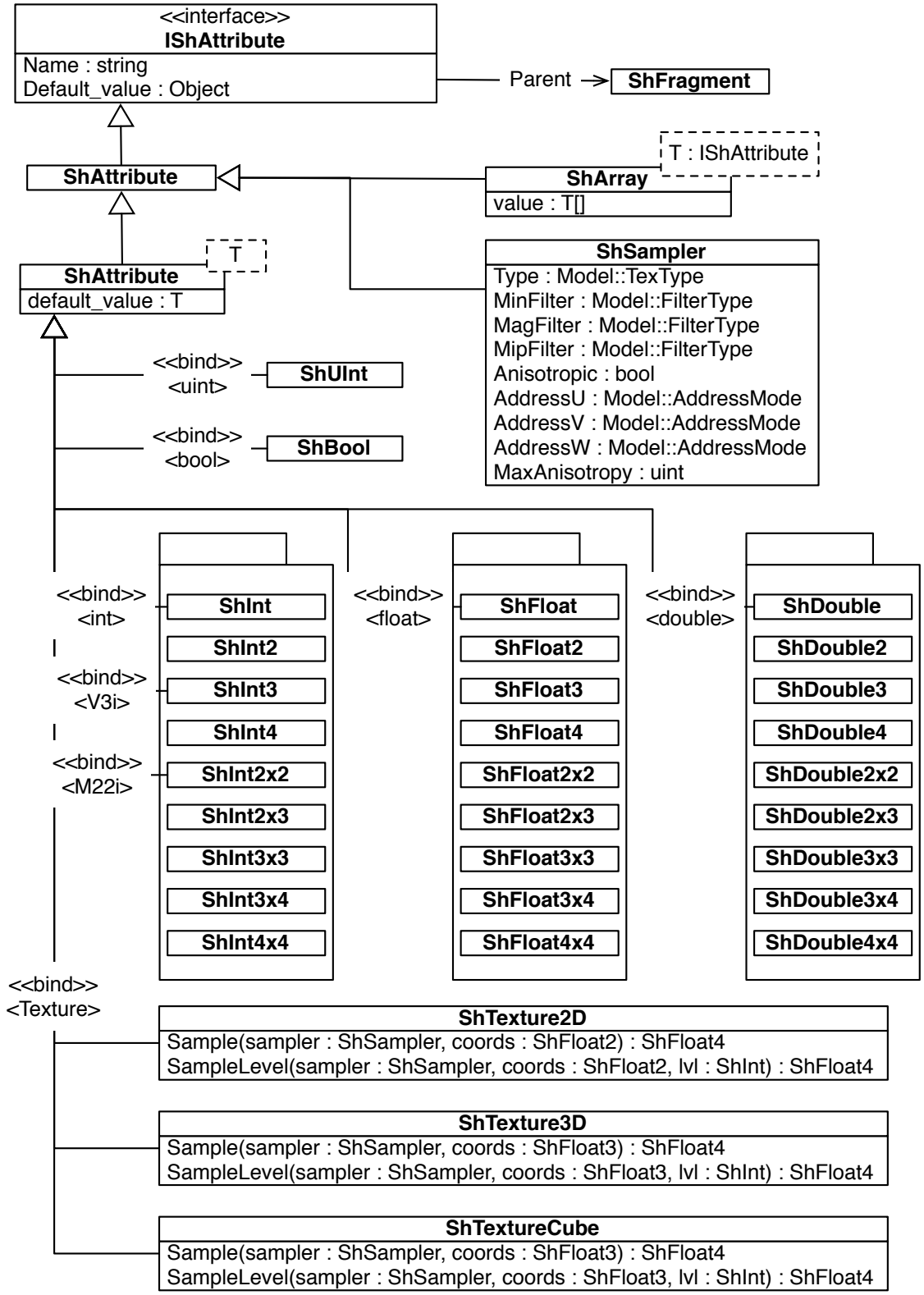
A.1.1 Effect and Fragments



A.1.2 Expressions

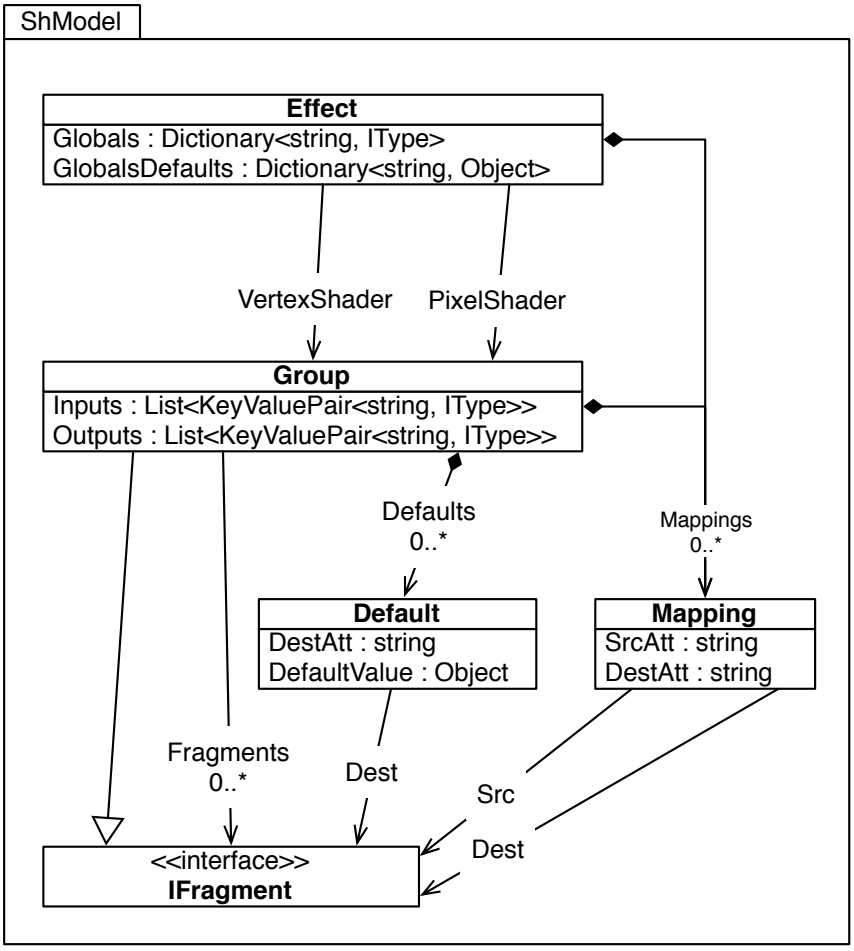


A.1.3 Attributes

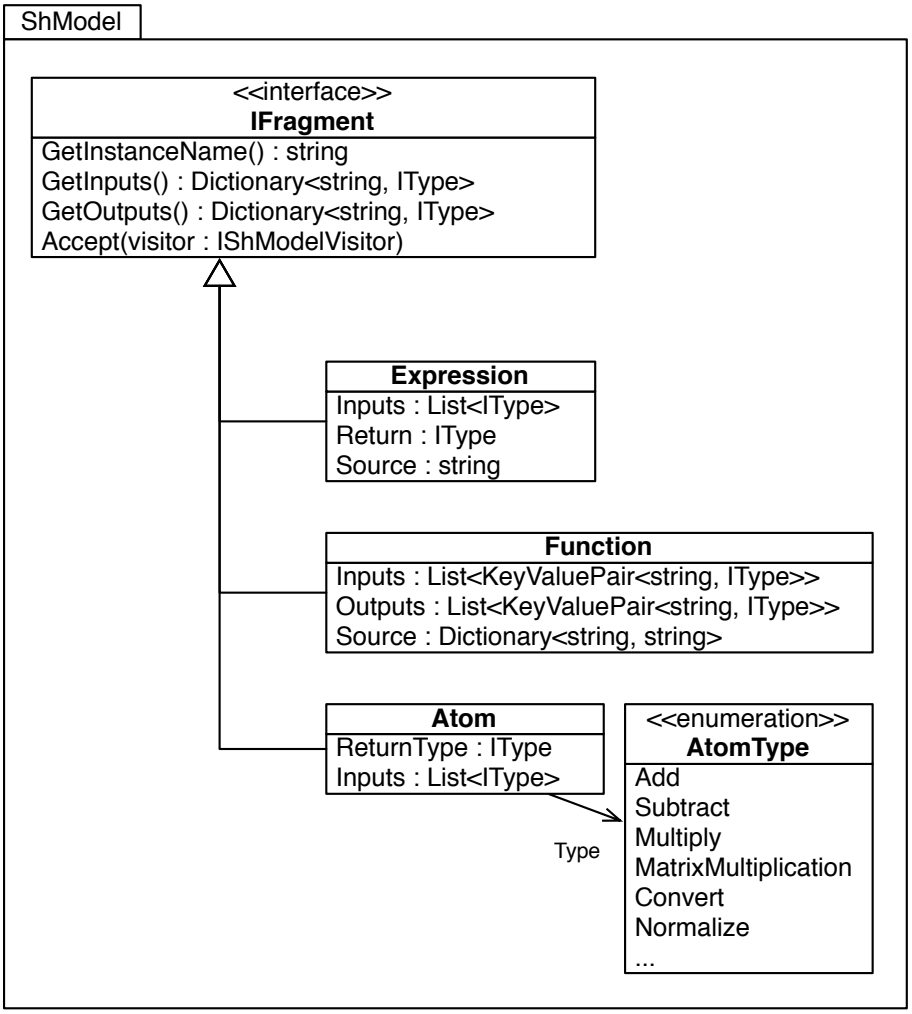


A.2 Semantic Model

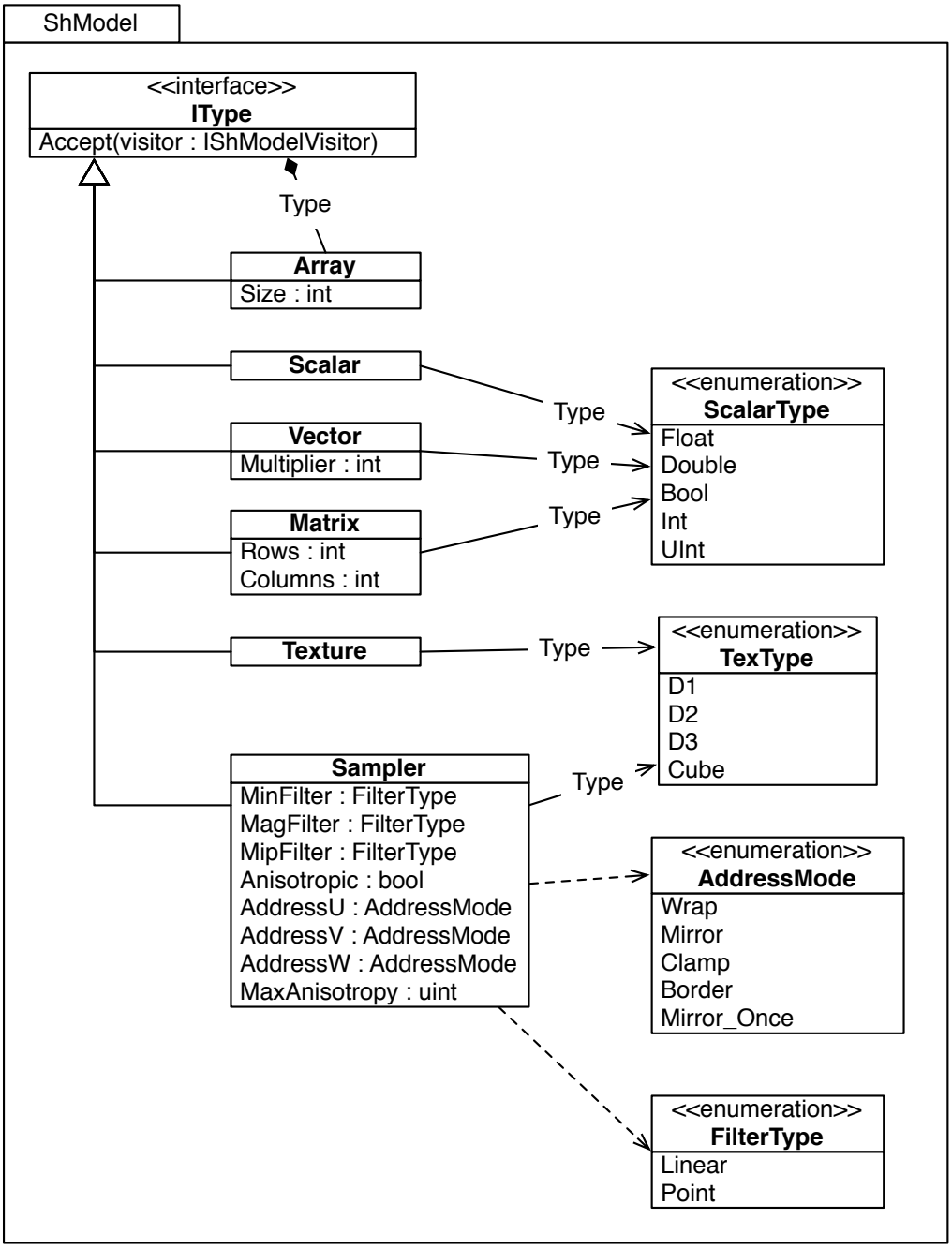
A.2.1 Effect and Group



A.2.2 Fragments



A.2.3 Types


ShModel
<<interface>>
IType
Accept(visitor : IShModelVisitor)
Type
Array
Size : int
Scalar
Vector
Multiplier : int
Matrix
Rows : int
Columns : int
Texture
Sampler
MinFilter : FilterType
MagFilter : FilterType
MipFilter : FilterType
Anisotropic : bool
AddressU : AddressMode
AddressV : AddressMode
AddressW : AddressMode
MaxAnisotropy : uint
Type
Type
Type
Type
Type
<<enumeration>>
ScalarType
Float
Double
Bool
Int
UInt
<<enumeration>>
TexType
D1
D2
D3
Cube
<<enumeration>>
AddressMode
Wrap
Mirror
Clamp
Border
Mirror_Once
<<enumeration>>
FilterType
Linear
Point

Glossary

Cg a shading language developed by Nvidia® in close collaboration with Microsoft® . 10, 12, 17, 52, 69

CPU Central processing unit. 2–4, 11, 14, 17, 37, 41, 48, 52, 56, 58, 67, 69, 76

GLSL OpenGL® Shading Language, created by OpenGL® ARB and further developed by Khronos Group. 10, 12, 18, 52, 69, 72

GPGPU General-Purpose computing on GPUs. 10

GPU Graphic processing unit. vii, 3, 9–11, 14, 17, 18, 28, 39, 45, 47, 48, 50, 66, 73, 76

HLSL High Level Shading Language, developed by Microsoft® for their Direct3D® API. 5, 10, 12, 13, 37, 40, 47–52, 65–67, 69, 76, 88, 89

Khronos Group not-for-profit member-funded industry consortium, focused on the creation of open standard, royality-free APIs. 10

OpenGL® ARB an industry consortium that goverened the OpenGL® specs. Superseded by Khronos Group. 10

VRVis Competence Center for Virtual Reality and Visualization. v, vii, 37

List of Figures

List of listings

Bibliography

[AMD] AMD® . Amd shader debugger. available from: http://developer.amd.com/tools/graphics-development/gpu-perfstudio-2/gpu-perfstudio-2-shader-debugger/ (accessed 2 December 2012).

[AMH02] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2002.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[Ana] Ananth Balasubramaniam. Brahma. available from: http://ananthonline.net/brahma/ (accessed 4 April 2012).

[AR05] Chad Anthony Austin and Dirk Reiners. Renaissance: A functional shading language. In *In Proceedings of Graphics Hardware 2005*, Graphics Hardware 2005. ACM, 2005.

[AW90] Gregory D. Abram and Turner Whitted. Building block shaders. *ACM SIGGRAPH Computer Graphics*, 24(4):283–288, September 1990.

[Bau07] Matthias Bauchinger. Designing a modern rendering engine, 8 2007.

[BSL+11] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100, October 2011.

[CHHE07] Chris Coleman, Kevin Harris, Anthony Hinton, and Anton Ephanov. Automated Shader Generation using a Shader Infrastructure. In *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, volume 2007, pages 1–9. NTSA, 2007.

[CNS+01] Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, and Pat Hanrahan. Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware. *System*, 2001.

[Coo84] Robert L Cook. Shade trees. In *ACM Siggraph Computer Graphics*, volume 18, pages 223–231. ACM, 1984.

[Ell04] Conal Elliott. Programming Graphics Processors Functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[FH11] Tim Foley and Pat Hanrahan. Spark : Modular , Composable Shaders for Graphics Hardware. In *ACM Transactions on Graphics (TOG)*, volume 30, page 107. ACM, 2011.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[FrW04] Niklas Folkegå rd and Daniel Wessllén. Dynamic code generation for realtime shaders. *Proceedings of SIGGRAD*, pages 11–15, 2004.

[GBD04] Frank Goetz, Ralf Borau, and Gitta Domik. An xml-based visual shading language for vertex and fragment shaders. In *Proceedings of the ninth international conference on 3D Web technology*, Web3D '04, pages 87–97, New York, NY, USA, 2004. ACM.

[GD06] Frank Goetz and Gitta Domik. Visual shaditor: a seamless way to compose high-level shader programs. In *ACM SIGGRAPH 2006 Research posters*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[HL90] Pat Hanrahan and Jim Lawsont. A Language for Shading and Lighting Calculations. *Computer*, 24(4):289–298, 1990.

[Id] Id Software. DOOM press release. available from: http://www.rome.ro/lee_killough/history/doompr3.shtml (accessed 17 August 2012).

[JFLC07] Peter Dahl Ejby Jensen, Nicholas Francis, Bent Dalgaard Larsen, and Niels Jørgen Christensen. Interactive shader development. In *Proceedings of the 2007 ACM SIGGRAPH symposium on Video games*, Sandbox '07, pages 89–95, New York, NY, USA, 2007. ACM.

[Kle] Klein, Strengert. Opengl glsl debugger. available from: http://www.vis.uni-stuttgart.de/glsldevil/ (accessed 2 December 2012).

[Kuc07] Roland Kuck. Object-oriented shader design. *Eurographics Short Papers*, pages 65–68, 2007.

[KW09] Roland Kuck and Gerold Wesche. A Framework for Object-Oriented Shader Design. *Assembly*, pages 1019–1030, 2009.

[LM] Erik Lindholm and Henry Moreton. A User-Programmable Vertex Engine. pages 149–158.

[MDP+04] Michael McCool, Stefanus Du Toit, Tberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 787–795. ACM, 2004.

[Mica] Microsoft® . Direct3d 11 graphics pipeline. available from: http://msdn.microsoft.com/en-us/library/ff476882.aspx (accessed 21 April 2012).

[Micb] Microsoft® . Direct3d 11.2 features. available from: http://msdn.microsoft.com/en-us/library/windows/desktop/dn312084%28v=vs.85%29.aspx (accessed 17 October 2013).

[Micc] Microsoft® . Direct3d compute shader overview. available from: http://msdn.microsoft.com/en-us/library/ff476331.aspx (accessed 20 April 2012).

[Micd] Microsoft® . Msdn debugging directx graphics. available from: http://msdn.microsoft.com/en-us/library/hh315751(v=vs.110).aspx (accessed 2 December 2012).

[Mice] Microsoft® . Msdn reference for hlsl. available from: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509638(v=vs.85).aspx (accessed 23 April 2012).

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68. Eurographics Association, 2002.

[MSPK06] M. McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. Abstract shade trees. *Proceedings of the 2006*, pages 79–86, 2006.

[Nvi] Nvidia® . Nsight - gpu development platform. available from: http://www.nvidia.com/object/nsight.html (accessed 28 August 2013).

[OKS03] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14. Eurographics Association, 2003.

[Per85] Ken Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3), 1985.

[Pha04] Matt Pharr. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 32. An Introduction to Shader Interfaces. Pearson Higher Education, 2004.

[PMT01] Kekoa Proudfoot, WR Mark, and S Tzvetkov. A real-time procedural shading system for programmable graphics hardware. *on Computer graphics*, 2001.

[SKE07] M. Strengert, T. Klein, and T. Ertl. A Hardware-Aware Debugger for the OpenGL Shading Language. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 81–88. Eurographics Association, 2007.

[TD07] Matthias Trapp and Jürgen Döllner. Automated combination of real-time shader programs. In *Proceedings of Eurographics*, pages 53–56, 2007.

[Tob11] Robert F. Tobler. Separating semantics from rendering: a scene graph based architecture for graphics applications. *Vis. Comput.*, 27(6-8):687–695, June 2011.

[Wika] Wikipedia. Microsoft® Direct3D. available from: http://en.wikipedia.org/wiki/Direct3D (accessed 19 April 2012).

[Wikb] Wikipedia. Moore's law. available from: http://en.wikipedia.org/wiki/Moore's_law (accessed 18 April 2012).

[Wikc] Wikipedia. Quartz composer. available from: http://en.wikipedia.org/wiki/Quartz_Composer (accessed 5 May 2012).

[Wikd] Wikipedia. Transistor count. available from: http://en.wikipedia.org/wiki/Transistor_count (accessed 13 April 2012).