Design and Implementation of a Shader Infrastructure and Abstraction Layer

Michael May[*]
Supervised by: Robert Tobler[†] Michael Schwärzler[‡]

VRVis Research Center

Abstract

With current GPUs being more powerful than the CPU in certain domains, shader programming has become more important than ever. Although the tool-chain and language features of shader languages have recently improved, they are still not as sophisticated as those of common general-purpose languages like C++, C# or Java. In order to benefit from these existing features in shader programming, a seamless integration into an application language would be favorable.

In this paper, an integration of shader development into C# is presented. An internal domain specific language was created that uses the tool-chain of C# and makes shader development part of the application language. Some of the benefits are shader type checks at C# compile time, use of the IDEs auto-completion feature for shaders and a flexible backend that can support the creation of different shader languages.

Keywords: domain-specific languages, shading languages, procedural shading, code generation, C#

1 Introduction

Shader development for programming the graphics pipeline is a key element in the creation process of a modern 3d application. Although the flexibility of shader languages and the available feature sets have increased drastically over the last few years, the handling of a large number of shaders is still a challenge. An application might have thousands of different effects for all the materials used in its different scenes, but only accesses a few at a time. This gives the challenges of both management and optimizations.

The traditional approach is to use *highly specialised shaders*, where each material is implemented by a shader with just the effects that are needed. This approach is optimized for runtime performance, but leads to a lot of code duplication. A different approach is the use of a so-called *Über-shader* – a big single shader that implements all needed effects and creates permutations either at compile (see Supershader [9]) or run-time. This omits code duplication, but might have lower run-time performance than the manually optimized code of the highly specialised shaders.

Both solutions become hard to maintain and to extend with a growing number of supported effects and permutations. This paper tries to tackle this challenges by presenting a design and implementation of a shader development infrastructure in C#. With the help of well-established software architecture patterns (see Section 3), code reusability, modularity and expandability is achieved. While the semantic model describing the operations of a shader in our framework is highly influenced by the shade tree concept (see Section 4), an internal domain-specific language (see Section 5) is used as an abstraction layer for the shader development, making the use of IDE features like auto-completion and type checks possible. A HLSL code generator produces legacy shader code from the iDSL (see Section 6). An example (see Section 7) demonstrates the capabilities of the presented framework.

2 Background

This work is based on the concepts of *shade trees* and *domain-specific languages*. Those roots and related work are presented in this section.

2.1 Shade Trees

Robert L. Cook proposed a flexible shading model, he called *Shade Trees* [2]. It describes a directed acyclic rooted graph, where nodes represent operations like a dot product and produce the final color in the root of the tree. For different parts of the shading process shade trees can be specified, e.g. for surfaces and light sources. A simple language was created to define a shade tree. This made shading more flexible, but it was still lacking higher control flow, like loops or conditional branching.

Pixel Stream Editor A more powerful language was introduced by Perlin for his Pixel Stream Editor called a *Pixel Stream Editing Language* [10]. It processes each

[*]may@vrvis.at
[†]rft@vrvis.at
[‡]schwaerzler@vrvis.at

pixel of an image and has functionality similar to the programming language C.

Shading Language Hanrahan and Lawson combined the ideas of Cooks modular shading and Perlins higher level shading language to create a language for Pixar's® RenderMan® and called it a *Shading Language* [5].

Shader Languages When hardware developers introduced programmable graphics cards, they called their languages *Shader Languages*. Three main hardware shader languages exist: HLSL for Microsoft's® DirectX, GLSL for OpenGL® and Cg from NVidia® .

In this paper we are talking about *Shader Trees*, giving credit to the roots in Cooks shade trees, but emphasizing its connection to modern shader languages.

2.2 Domain-Specific Languages

A Domain-Specific Language (DSL) is a programming language of limited expressiveness focused on a particular domain [4]. Shader languages are such a languages with graphics processing as their domain. Two types of DSL can be distinguished:

External Domain-Specific Languages (eDSL) are independent languages with their own tool-chain. They can be specially tailored to their needs, but can only rely on tools created for them. Examples of an eDSL are shader languages like HLSL, SQL and XML.

Internal Domain-Specific Languages (iDSL) are embedded into a general-purpose language. An iDSL is valid code in its host language and can therefore use the existing development tool-chain. This means that the iDSL can use the debugger or type system of its host, but also that it has to operate inside its limitations. Functional languages like LISP have a long history of using iDSLs, but also modern main stream languages seem to rediscover their benefits, like C# with LINQ.

2.3 Extending Shader Languages

Different projects extended shader languages with functional or object-oriented designs by means of external or internal DSLs.

Functional Programming It has been argued that shaders map well on functional languages, with side-effect-free shader stages working parallelly over a stream of data, reminding of pure functions over lists. *Renaissance* is a functional approach for a shader language in terms of an eDSL [1]. *Vertigo* [3] on the other side embeds shader into the pure functional programming language Haskell and facilitates partial evaluation and symbolic optimization. An example for the good optimization is the automatic avoidance of multiple normalizations of a vector, facilitating expression rewriting.

Although functional programming has a lot of advantages and a long history of high level language features, it is not considered as a popular main stream application model. With shader languages feeling like a procedural language, there might be a problem with alienating shader programmers.

Object-Oriented Programming Kuck and Wesche introduced object-oriented design into existing shader languages [6, 7] relying on the shader compiler to optimize the added complexities. The introduction of shader interfaces introduced object-oriented programming for dynamic linking natively.

McCool, Qin and Popa developed a system that integrates shader into C++ and called it *Sh* [8]. The shader is programmed as a sequence of function calls, where the variables are smart reference-counting pointers that create a parse tree. Preprocessor macros are used to make the syntax cleaner.

Although shaders in *Sh* benefit from the C++ integration, C# offers additional features in the language and tool-chain (e.g. Reflection and IntelliSense) that might benefit shader development.

3 Concept

The design concept of the proposed framework is based on the *Model-View-Controller* pattern (MVC), that uses a shader tree as an internal presentation (see Section 4).

The MVC is a software architecture pattern that separates user input (the controller), data storage (the model) and representation of data (the view) as shown in Figure 1. By supplying an individual structure for shader definition (iDSL/controller) and one for processing (semantic model), they can each be optimized for their specific task.

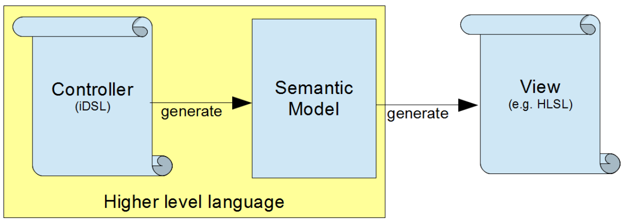


Figure 1: Concept based on the MVC pattern. A shader is defined in the iDSL, which generates a semantic model for further processing, that is mostly shader code in the end.

The conceptual layout in regard of MVC is:

Controller The iDSL is C# code, that gives the feeling of coding in a shader language. The defined shader code is put in special classes for structuring and defining shader stages. Connecting the shader to the application is straight forward by assigning the right variables (see Section 5).

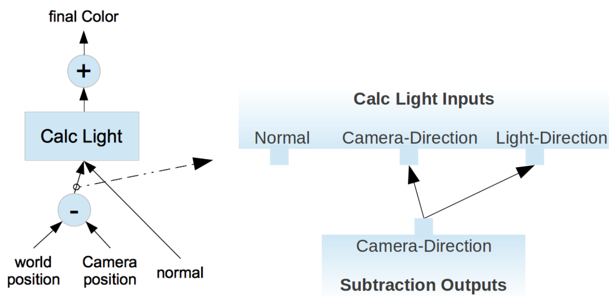
Model While the domain-specific language has defining shaders and convenience for the programmer in mind, the semantic model is meant for processing. It is a shader tree, that represents the functionality of the shader that was defined in the controller (see Section 4).

View The semantic model is translated into a concrete shading language, that is than compiled by a shader compiler and sent to the graphics processor (see Section 7).

4 Semantic Model

The semantic model is a shader tree implementation and is the *model* in the MVC design. Its main functionality is to represent the defined shader for further processing (see Section 6).

Shader Tree A shader tree is a rooted directed tree with the nodes being shader operations (see Figure 2a) and the edges being the mappings between inputs and outputs (see Figure 2b).



(a) Shader tree (b) Mapping

Figure 2: A Shader tree consists of operations or shader fragments as nodes (see (a)) and the edges are input/output mappings between those fragments (see (b)).

There are four types of shader nodes (also called shader fragments, see Figure 3). *Atom* and *group* fragments are the basic building stones for a shader, while *expression* and *function* fragments are used to introduce legacy shader code for prototyping.

- *Atom* This node is used for all basic operations that a shader supports. This can be dot product, matrix multiplication or even swizzle operators. In Figure 2a the + and - are atom fragments.
- *Group* This node can group together other fragments, even other group nodes, to structure code and facilitate code reuse. In Figure 2a the `Calc Light` fragment might be a group node.
- *Expression* This is a node to embed legacy shader code for prototyping (see Section 5.3). It consists of a short piece of code with only one return value.
- *Function* This is a node to embed legacy shader code for prototyping (see Section 5.3). It can be several lines of code long and can have multiple outputs. In Figure 2a the `Calc Light` fragment could be a function node.

5 Internal DSL

The iDSL is the interface to the programmer to define a shader and represents the *controller* in the MVC design, so usability is the prime directive. Based on this definition the semantic model is created.

To define the later semantic model in the iDSL, the iDSL has to have the same information stored as the model, so it is by itself a shader tree. There are different nodes for different purposes (see Figure 3 and Section 5.1, Section 5.3 and `ShGroup` in Section 5.2), but they are all based on `ShFragment`. The edges or connection information between the nodes are handled by `ShAttribute`. The class `ShEffect` binds together code for different shader stages to one complete shader (see Section 5.2).

ShAttribute All inputs and outputs of fragments are `ShAttributes`. They are used to connect those fragments with each other and later create mappings of the semantic model (see Figure 2b). As outputs they hold a reference to their parent fragment and are then used as inputs to connect nodes.

Additionally they are used to set default values or new values during the runtime of the shader from the main application. Therefore special variations for different types exist of the `ShAttribute`.

```
interface IShAttribute;
abstract class ShAttribute:IShAttribute;
abstract class ShAttribute<T>:ShAttribute;
class ShTexture2D:ShAttribute<Texture>;
class ShTexture3D:ShAttribute<Texture>;
class ShTextureCube:ShAttribute<Texture>;
class ShArray<T> : ShAttribute
    where T : IShAttribute, new();
class ShSampler : ShAttribute;
class ShBool : ShAttribute<bool>;
class ShInt : ShAttribute<int>;
class ShFloat3 : ShAttribute<V3f>;
```

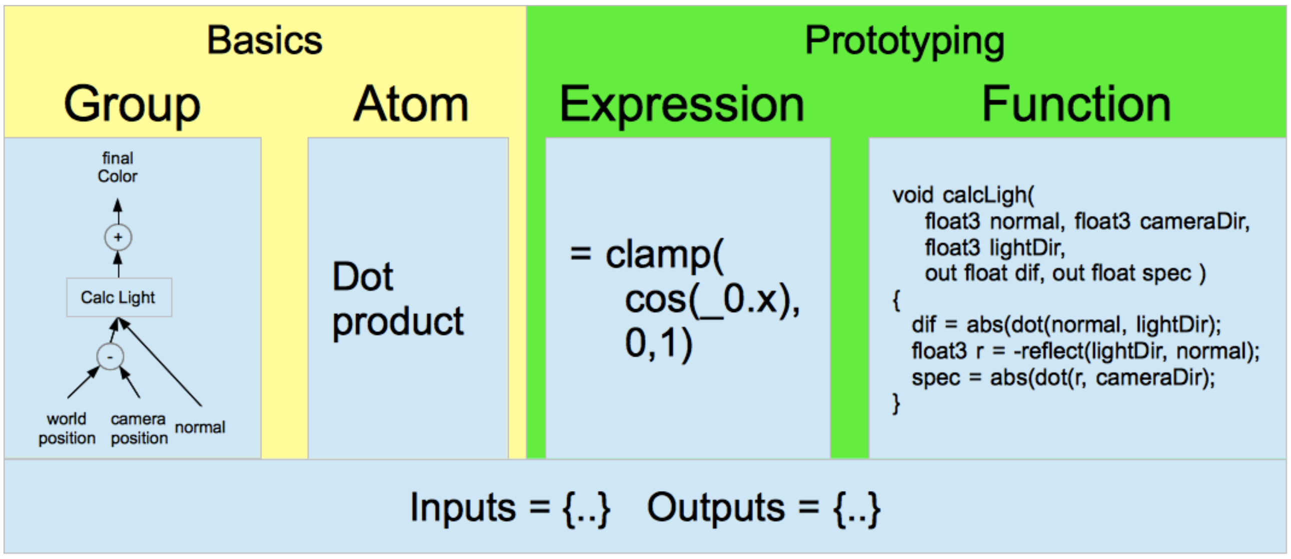

Basics
Prototyping
Group
Atom
Expression
Function
final Color
+
Calc Light
-
world position
camera position
normal
Dot product
= clamp(cos(_0.x), 0,1)
void calcLigh(float3 normal, float3 cameraDir, float3 lightDir, out float dif, out float spec) { dif = abs(dot(normal, lightDir); float3 r = -reflect(lightDir, normal); spec = abs(dot(r, cameraDir); }
Inputs = {..}
Outputs = {..}

Figure 3: The most basic fragments are the *Atom* for all basic operations and the *Group* for modularization. To introduce legacy shader code for prototyping there is also an *Expression* and a *Function* node.

```
class ShFloat4x4 : ShAttribute<M44f>;
...
```

5.1 Programming

Writing a shader program in the iDSL looks like programming in any other programming language. Variables are of the type `ShAttribute` and connecting two nodes is done by calling the target like a function with the outputs of the sources as inputs. Every such call generates a fragment instance as node of the shader tree.

ShAtom The following code shows some examples of shader code in the iDSL. These are all functions implemented in one of the subclasses of `ShAttribute` which create `ShAtom` nodes.

```
var bumpTexTS =
    texture.Sample(sampler,inPosOS.XY/50);

var normWS =
    new ShFloat3(bumpTexTS.XY,inNormOS.Z)
    .Mul((ShFloat3x3)inMTrafoTI)
    .Normalize();

var reflVec=vecVer2Cam.Reflect(normWS);

var transparency =
    1.5f - normWS.Dot(vecVer2Cam).Abs();
```

5.2 Structuring

A `ShEffect` represents a complete shader with all needed shader stages. Each shader stage is a `ShGroup`, which holds further iDSL shader code.

ShGroup To reduce code duplication the `ShGroup` can encapsulate several lines of iDSL code to be called like one fragment. An instance of the ShGroup is created inside iDSL code by invoking its `Call` method with the inputs as parameters. Outputs are defined as class fields. The encapsulated iDSL shader code is defined inside the `Call` method. The code sets output values by assigning to the appropriate output fields of the created `ShGroup` instance. The registration of input values at the end is also important to be able to build the semantic model.

```
class TestShaderStage : ShGroup
{
  public ShFloat4 ReturnValue;

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

     group.ReturnValue =
        inPosWS * inNormWS;

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

ShEffect `ShEffect` binds all the code together for a complete shader. The base class has defaults defined for

Globals and inputs/outputs from different shader stages, but each new effect can define their own values. The Link method creates instances of the shader stages, which are ShGroups and declares connections between the stages and to/from the pipeline.

```
class SimpleNormalShader : ShEffect {
  public Globals Global;
  public VertexInputs VertexInput;
  public VertexOutputs VertexOutput;
  public PixelOutputs PixelOutput;

  public override void Link()
  {
    var vShader = vertexShader.Call(
      VertexInput.Positions,
      VertexInput.Normals,
      Global.ModelViewProjTrafo);
    var fShader = pixelShader.Call(
      vShader.outNormOS);

    VertexOutput.Position =
      vShader.ReturnValue;
    PixelOutput.ImageOutput =
      fShader.ReturnValue;

    Init(vShader, fShader);
  }
  public class vertexShader : ShGroup;
  public class pixelShader : ShGroup;
}
```

5.3 Prototyping

These nodes are meant for prototyping and not for final production code, because they introduce legacy shader code into the system and therefore limit its usability to one shader language. They still can be useful to test existing shader code, that is finally rewritten with ShAtom nodes for the final production code.

ShExpression An expression is used to 'inline' a short piece of shader code. It is not defined in which shader language this code is written. The return type must be specified, but the input types are determined by the given inputs. The legacy shader code is defined in a string, where the output is the return value of the expression and the inputs are represented by using an underscore and the position of the input as identifier.

```
var randomTex =
  ShExpression<ShFloat>.Call(
  "clamp(cos(_0.x * 2 + _1.x),0,1)"
  , noise.ReturnValue, inPosOS);
```

ShFunction To test shader code that has multiple outputs or more than one line, a ShFunction is used. Outputs are defined as fields and inputs as parameters of the Call method. In this method, a new node instance has to be created, and the legacy code and inputs have to be registered. Also, the type of shader language is specified to make early compatibility checks. To use the defined function, only the Call method has to be called in the shader definition.

```
class NoiseFromStatic3D : ShFunction {
  public ShFloat ReturnValue;

  public static NoiseFromStatic3D Call(
    ShTexture3D tex,ShSampler sampler,
    ShFloat3 texCoord,ShInt numOfSamples)
  {
    return CreateInstance
      <NoiseFromStatic3D>
      (Tokens.HLSL, HLSLFuncCode, tex,
      sampler, texCoord, numOfSamples);
  }

  private static readonly string
  HLSLFuncCode =
@"{
  float perl = 0;
  for(int i=0; i<(numOfSamples-1); i++)
  ...
  return perl;
}";
  }
```

6 Processing

The *Visitor* pattern is used for processing the *semantic model* (see Figure 4). This is a software design pattern that separates an algorithm from the object it operates on. It keeps the model simple and extending can easily achieved by adding new visitors. Such processing tasks can be e.g. optimizations, error checking and code generation.

HLSL Code Generation Each ShFragment and ShEffect can generate its corresponding semantic model. The ShAttributes are analysed for the connection information and namings are gathered with reflection and passed through to the model. The iDSL is parsed depth-first in pre-order.

The visitor creates HLSL code based on the semantic model (See example in Figure 5). Effect model creates an HLSL effect, Atom models are converted based on a translation table and Expression models get inlined. Group and Function models become HLSL functions, but their use gets logged, so that only one HLSL function per model and per instance is created.

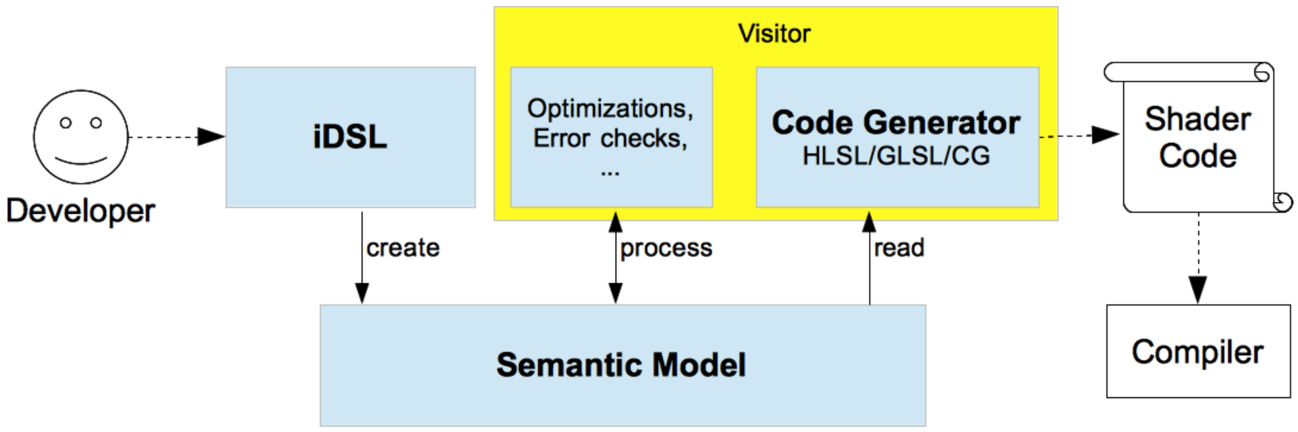


Figure 4: Processing on the semantic model is based on the Visitor Pattern, to keep the model simple, but also extendible.

In the iDSL a programmer can name fragment instances and this will be used to name its output variables in the HLSL code. All other variables become a default name with a sequential number as identifier. The Example Figure 5 shows named and unnamed instances.

7 Results

To demonstrate the capabilities of the proposed framework, four examples have been implemented and compile times have been taken to test the performance.

7.1 Example

To test the implementation four shaders have been implemented (see Test-Fountain.cs as well as the generated .fx HLSL files in the accompanying additional material) to demonstrate procedural textures, animated vertex displacement, animated particle system with instancing and multi texturing. The test scene is a water fountain on a grass hill (see Figure 6).

Sprinkle Shader This shader generates drops of water coming out of the top of the fountain by using a *particle system with instancing*. The simulation of the water drops is done on the CPU and produces one transformation per drop. One model of a water drop is sent to the GPU, where it is duplicated(instanced) and transformed by the numbers of transformation from the simulation.

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. *Vertex displacement* is done in the vertex shader by adjusting the position by the normal map. In the pixel shader reflections of an environment map and transparency is computed based on the normal adjusted by the normal map.

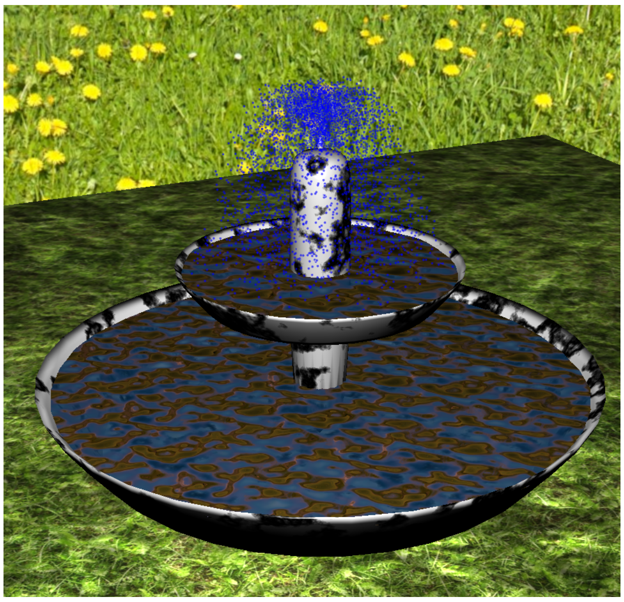


Figure 6: Implemented example to demonstrate procedural textures, animated vertex displacement, animated particle system with instancing and multi texturing.

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 easily be integrated. The *marble shader* generates a *procedural texture* with perlin noise based on a random texture. In the *grass shader* a 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. This *multi texturing* hides artefacts created by patterns.

```
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 5: An iDSL converted to HLSL shader code. HLSL code pieces are generated from the iDSL with the same colour.

Overview Following table shows an overview of the used shadertree fragments in the examples. Typically two groups are used at least for the vertex- and fragment shader, but they might as well be any other type of fragment.

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

7.2 Code Analysis

To evaluate the presented framework the effects from the examples where also implemented directly in HLSL to compare with the generated versions from the iDSL.

Complexity The following table compares the shader examples written in the iDSL with manual 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 HSL code.

	lines code			lines overhead		
Shader	iDSL	Man	fac	iDSL	Man	fac
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 involves less than one and a half as much overhead.

The smaller the fragment the more overhead it has with the smallest function fragment having four times the overhead in the iDSL compared to the HLSL code. These smaller fragments 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, but the sprinkle shader.

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 finaly the HLSL shader compiler (GPU).

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 we provide a much more convenient programming model, the overall compile time impact is moderate: The additional overhead introduced by our 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).

Run Time The rendering time needed for a generated shader was compared to a manual optimized one with no noticeable difference. The presented examples where used, but neither them 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.

Debugging The proposed framework introduces additional possibilities for debugging shaders:

Host language The development environment of C# Visual Studio supports finding errors in the iDSL already while writing code, e.g. by highlighting type errors or typos. 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.

Shader language The generated code is legacy shader code and therefore it can be debugged as 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.

8 Conclusions

In the presented work, a framework for integrating shader development into the host language C# has been proposed, facilitating the generation, management and re-usability of shader code. This is achieved by embedding the concept of shader trees into an internal domain specific language, allowing well-know and often-used IDE features like auto-completion and type safety to be used in the otherwise tiresome generation of HLSL code.

Although this approach introduces a small overhead in terms of the overall compile time, the advantages gained from this convenient programming model may prevail in many situations where rapid and reliable shader development is of importance.

Future improvements in the proposed system could be the support of other shader languages (GLSL, CG, WebGL), basic control structures (branching and looping) and further shader pipeline features (stages like geometry or tesselation shaders, or multipass rendering).

References

[1] C.A. Austin. *Renaissance: A Functional Shading Language*. Iowa State University, 2005.

[2] Robert L Cook. Shade Trees. In *ACM Siggraph Computer Graphics*, pages 223–231. ACM, 1984.

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

[4] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series. Pearson Education, 2010.

[5] Pat Hanrahan and J Lawson. A Language for Shading and Lighting Calculations. *ACM SIGGRAPH Computer Graphics*, 24(4):289–298, 1990.

[6] Roland Kuck. Object-Oriented Shader Design. *Eurographics Short Papers*, pages 65–68, 2007.

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

[8] 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.

[9] Morgan McGuire. The SuperShader. In Wolfgang Engel, editor, *ShaderX4*, chapter 8.1, pages 485–498. Charles River Media, Inc., 2005.

[10] Ken Perlin. An Image Synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3), 1985.