Cross-Compiling Shading Languages

Masterthesis

Lukas Hermanns
KOM-M-0598

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Elektrotechnik
und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation
Prof. Dr.-Ing. Ralf Steinmetz

Cross-Compiling Shading Languages
Masterthesis
KOM-M-0598

Submitted by Lukas Hermanns
Day of submission: October 19, 2017

Consultant: Prof. Dr.-Ing. Ralf Steinmetz
Supervisor: Robert Konrad

Technische Universität Darmstadt
Fachbereich Elektrotechnik und Informationstechnik
Fachbereich Informatik (Zweitmitglied)

Fachgebiet Multimedia Kommunikation (KOM)
Prof. Dr.-Ing. Ralf Steinmetz

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterthesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 19. Oktober 2017 Lukas Hermanns

Abstract

Shading languages are the major class of programming languages for a modern mainstream Graphics Processing Unit (GPU). The programs of those languages are called “Shaders” as they were originally used to describe shading characteristics for computer graphics applications. To make use of GPU accelerated shaders a sophisticated rendering Application Programming Interface (API) is required and the available rendering APIs at the present time are *OpenGL*, *Direct3D*, *Vulkan*, and *Metal*. While Direct3D and Metal are only supported on a limited set of platforms, OpenGL and Vulkan are for the most part platform independent. On the one hand, Direct3D is the leading rendering API for many real-time graphics applications, especially in the video game industry. But on the other hand, OpenGL and Vulkan are the prevalent rendering APIs on mobile devices, especially for Android with the largest market share.

Each rendering API has its own shading language which are very similar to each other but varying enough to make it difficult for developers to write a single shader to be used across multiple APIs. However, since the enormous appearance of mobile devices many graphics systems are reliant on being platform independent. Therefore, several rendering technologies must be provided as back ends. The naive approach is to write all shaders multiple times, i.e. once for each shading language which is error-prone, highly redundant, and difficult to maintain.

This thesis investigates different approaches to automatically transform shaders from one high-level language into another, so called “cross-compilation” (sometimes also referred to as “trans-compilation”). High-level to high-level translation is reviewed as well as algorithms with an Intermediate Representation (IR) such as Standard Portable Intermediate Representation (SPIR-V). We are focusing the two most prevalent shading languages, which are firstly OpenGL Shading Language (GLSL) and secondly DirectX High Level Shading Language (HLSL), while Metal Shading Language (MSL) is only briefly examined. The benefits and failings of state-of-the-art approaches are clearly separated and a novel algorithm for generic shader cross-compilation is presented.

Acknowledgment

First of all, I would like to thank my family for their support during my entire study. Through their support, they made many of the circumstances beside my study much easier. I would also like to thank Robert Konrad, who gave me the opportunity to base my thesis upon a long-term project in the research field of cross-platform development for real-time graphics applications. Finally, I would also like to thank Marko Pintera for his contribution to the XShaderCompiler project.

Contents

1 Introduction

1.1 Overview

In this thesis a novel method for cross-compiling shading languages is presented and compared to other state of the art approaches. The main focus is on the two mainstream shading languages HLSL and GLSL for real-time graphics applications.

HLSL is the shading language of the Direct3D rendering API, while GLSL and its subtly modified dialect OpenGL ES Shading Language (ESSL) is the shading language for the OpenGL, OpenGL ES, and WebGL rendering APIs. There is also the modern shading language called “Metal” (or rather MSL) that is supported by the correspondent rendering API, but due to its limitation for products by Apple Inc. and the alternative of ESSL it is only briefly examined. Although Direct3D is only available on a limited set of platforms as well, more precisely on platforms provided by Microsoft Corporation, its large distribution in the video game industry makes HLSL a major candidate for cross-compilation.

Although HLSL and GLSL have the same purpose and are quite similar in many ways, there are also lots of differences in both syntax and semantics. Even a very primitive shader program like shown in Listing 1.1 must be translated to something that looks quite different as shown in Listing 1.2, or vice versa. These two shaders both only pass the input vertex coordinate to the fragment shader.

Listing 1.1: HLSL

```
1 struct v2f {
2     float4 position : SV_Position;
3 };
4 v2f VS(float4 coord : COORD) {
5     return (v2f)coord;
6 }
```

Listing 1.2: GLSL

```
1 #version 130
2 in vec4 coord;
3 void main() {
4     gl_Position = coord;
5 }
```

This thesis covers different approaches to allow those shaders being written only once and to be used on multiple rendering APIs. These approaches reach from heavy macro meta-programming over simple parsers to full shader compilers.

1.2 Motivation

Platform independence is a very important quality for a broad range of applications. This applies to end user applications that are meant to run on a variety of desktop and mobile platforms such as Windows, macOS, GNU/Linux, Android, and iOS as well as to middleware software that composes the foundation of such end user programs.

In the situation of real-time graphics applications this requirement has changed in the recent years. Before the enormous appearance of mobile phones with 3D hardware acceleration (simply known as “smartphones”), the leading shading language was HLSL and its (now obsolete) dialect C for Graphics (Cg), which was developed by NVIDIA together with Microsoft [23, 34]. This was the case for at least the video game industry where HLSL (or Cg) was supported on the major gaming platforms: Microsoft® Windows, Microsoft® Xbox® 360, and Sony™ PlayStation® 3. In the field of research, GLSL was the prevalent shading language, due its platform independence on desktop computers. Though, since the market share of smartphones, especially those with Google Android and Apple iOS that do not support HLSL at all, has grown rapidly, the importance to support multiple shading languages in graphics applications has increased.

Because the naive approach of translating shader programs by hand is time consuming, error-prone, and hardly maintainable when shaders are altered or extended, an automatic translation of these pro-

grams is highly necessary. Since this problem is not new to the development process of platform independent rendering systems, many different approaches and solutions exist throughout the web and technology community. However, lots of these solutions are tied to a specific project or needs of a company, some are very limited in the sense of source language support, yet other approaches only support outdated versions of a shading language, and some cross-compilers depend on a platform specific tool and therefore restricting their benefit of platform independence.

One of the most relevant projects for shader cross-compilation is the GLSL reference compiler named glslang together with the SPIRV-Cross framework by The Khronos Group [22, 3]. This framework is primarily deployed to compile GLSL code into SPIR-V to be used for the Vulkan rendering API. Since glslang also has a front-end for HLSL, and SPIRV-Cross has multiple back-ends to produce high-level GLSL, HLSL, and MSL it is currently one of the most advanced shader cross-compilers. Therefore, the compiler framework by The Khronos Group is examined in more depth and compared to our novel method for cross-compiling shading languages for which an implementation exists in form of the XShaderCompiler project [14].

1.3 Outline

Here is a brief outline of the following chapters:

Chapter 2: Background
: Gathers the fundamental terms and concepts of shaders, shading languages, and compiler theory. There is a quick introduction into shaders including a simple visual example. The example is supplemented with a code review for all major shading languages of real-time graphics. Furthermore, the definitions and basics of formal languages are specified and examined. In the end, the theoretical structure of a compiler is examined.

Chapter 3: Related Work
: State of the art proceedings for cross-platform shader development is presented in this chapter. Not only other cross-compilers are reviewed but also entirely different approaches to make shaders available for multiple rendering systems: From naive approaches like macro expansion to novel algorithms involving dedicated shader compilers with high-level assembler concepts.

Chapter 4: Concept
: In this chapter the major differences between the two most widely used shading languages are analyzed in detail. The chapter is separated into many more or less independent sections. Each section describes a distinct topic where a sophisticated translation between the languages is necessary. There are also one or more examples for each section to illustrate the general idea of the translations. The chapter also investigates where compromises in the translation are inevitable. A summary of the most important differences is highlighted supplementary at the end of the chapter.

Chapter 5: Implementation
: The implementation details of a novel algorithm for shader cross-compilation are presented. Various details from the previous chapter are analyzed in more depth and also in relation to a practical shader compiler. Several proposals for adequate simplifications in the compiler implementation are presented, too.

Chapter 6: Results and Discussion
: The results of the novel algorithm are compared to those of other state of the art approaches as well as a hand written translation which is assumed as *ground truth*. Several shader examples are considered from both small sized shaders as well as large production shaders.

Chapter 7: Conclusion and Future Work

Summarizes the features of the novel method and clearly separates its benefits and failings compared to previous work. A conclusion for the use of this method is also formulated. Prospective ideas for further improvements are proposed. A few possibilities to extend a shading language specifically for the purpose of cross-compilation are presented in the appendix, too.

2 Background

Before we can look at *state of the art* methods for shader cross-compilation, we need to consider some fundamentals about shaders and compilers in general.

2.1 Shaders

Shading languages have their origin in Pixar's RenderMan shading language [11] and their programs are called "Shaders" as they were originally used to describe shading characteristics for computer graphics. Modern GPUs are no longer restricted to shading characteristics and their massively parallel execution units can be utilized for General Purpose Computing on Graphics Processing Units (GPGPU), such as physics and fluid simulations [27], but also statistical analysis and deep machine learning algorithms [48, 35]. The name "shading language", however, remained for GPU-based programs.

The shading languages we are focusing on in this thesis have several types of shaders, each for a certain stage within the graphics pipeline, whereas the shader type for GPGPU has its own encapsulated pipeline. All these types of shaders can be outlined in the following table:

Type	**Remarks**
Vertex Shader	Vertex processing such as world coordinate and normal vector transformation, and projection into screen space. This shader is often used for animation, but also to simply pass the input data on to the next shader stage.
Tessellation Control Shader (*also* Hull Shader)	Controlling tessellation of output patches. Can be used to let the GPU generate many geometric primitives (i.e. Points, Lines, or Triangles) out of a patch of up to 32 control points.
Tessellation Evaluation Shader (*also* Domain Shader)	Evaluates the input patch from the previous shader stage. Can be used to apply displacement mapping by transforming the tessellated primitives by a texture buffer for instance.
Geometry Shader	Generates or discards the final geometric primitives and passes them on to the rasterizer.
Fragment Shader (*also* Pixel Shader)	Generates or discards the final pixel fragments that might present a pixel on the screen. This is commonly the only shader stage that performs actual 'shading'.
Compute Shader	General purpose shader stage within its own encapsulated shader pipeline.

In the simplest form, a modern rendering system commonly needs at least a vertex- and a fragment shader to display something on the screen. Algorithm 1 illustrates the idea of a simple vertex- and fragment shader to render a basic *Lambertian Reflectance Model* [28, 36] for diffuse lighting. M_{wvp} specifies the *world-view-projection* matrix, to transform a coordinate from *object space* into *projection space*. M_w specifies the *world* matrix ($\in \mathbb{R}^{3\times3}$), to transform a normal vector from *object space* into *world space*, but without translation. V, N, and C specify the vertex coordinate, normal vector, and color respectively, the vector index *v2f* specifies the "vertex-to-fragment" exchange data, and the vector indices *in* and *out* specify the input and output data respectively.

In an optimal scenario of parallel computation, the vertex shader is then executed simultaneously for each vertex of the input model, and the fragment shader is executed simultaneously

Algorithm 1 Simple Vertex and Fragment Shader

1: **procedure** VERTEXSHADER($M_{wvp} \in \mathbb{R}^{4\times4}; M_w \in \mathbb{R}^{3\times3}; V_{in}, V_{v2f} \in \mathbb{R}^4; N_{in}, N_{v2f} \in \mathbb{R}^3$)
2: $V_{v2f} \leftarrow M_{wvp} \times P_{in}$ ▷ Transform coordinate from object- to projection space
3: $N_{v2f} \leftarrow M_w \times V_{in}$ ▷ Transform normal vector from object- to world space
4: **procedure** FRAGMENTSHADER($C_{diffuse}, V_{v2f}, C_{out} \in \mathbb{R}^4; N_{v2f}, L_{in} \in \mathbb{R}^3$)
5: $C_{out} \leftarrow C_{diffuse} \times \max\left\{0, ||N_{v2f}||_2 \cdot ||L_{in}||_2\right\}$ ▷ Compute diffuse lighting with Lambert factor

for each fragment. Therefore, a shader can be considered to be something like a formula to describe the resulting characteristic, and this formula is applied to each vertex or fragment independently. An overview of the major stages in a graphics pipeline is illustrated in Figure 2.1. While the *red* stages are so-called *fixed function* stages that can only be configured by a straightforward set of options, the *blue* stages are so-called *programmable* stages or rather the shader stages itself. Some shader stages are optional, like the tessellation shader stages for instance. Figure 2.2 illustrates an example implementation of the shader of Algorithm 1, with a sphere model constructed out of thousands of tiny triangles.

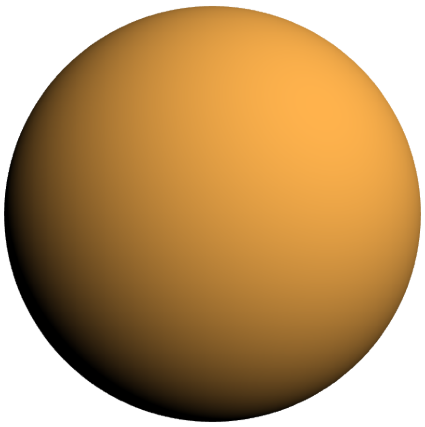


Figure 2.2: Lambertian Reflectance Model for diffuse lighting. $C_{diffuse} = (1, 0.7, 0.3, 1)^T$; $L_{in} = (1, 1, 3)^T$.

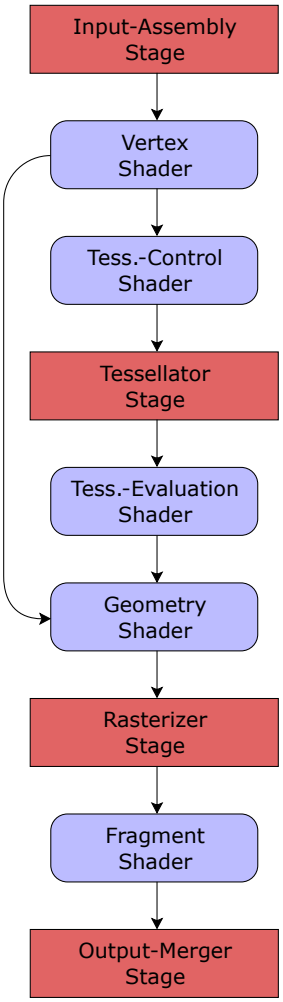


Figure 2.1: Graph of a graphics shader pipeline.

Most shading languages are derivations of the C programming language or rather a superset of the subset of C [45, 6]. Except MSL which is a superset of the subset of C++14 [15]. Therefore, at least the syntax of all shading languages is quite similar, especially the declaration of control flow like the well-known `for`-loop and `if`-condition statements. They also have the same limitations in common, one of which is the lack of *function recursion*, due to the missing stack and the parallel nature of GPUs. And *pointers* to global memory, which are a basic language construct in the C programming language, are not supported in GLSL and HLSL, too. Note that the GPGPU languages OpenCL and CUDA are exceptions in the case of pointers and other semantics [54, 47], but they are beyond the scope of this thesis.

Alongside all similarities between the shading languages we are targeting, one of the major difference is the input and output semantics. We now consider the example of the basic diffuse shading from above again, and look at a proposed implementation for GLSL (Listings 2.1, 2.2), HLSL (Listings 2.3, 2.4), and Metal (Listing 2.5):

Listing 2.1: GLSL: `diffuse.vert`

```
#version 460

uniform mat4 wvpMatrix;
uniform mat3 wMatrix;

layout(location = 0) in vec4 V_in;
layout(location = 1) in vec3 N_in;

out vec3 N_v2f;

// Vertex shader entry point
void main() {
  // Transform coordinate/normal vector
  gl_Position = wvpMatrix * V_in;
  N_v2f       = wMatrix   * N_in;
}
```

Listing 2.2: GLSL: `diffuse.frag`

```
#version 460

uniform vec4 C_diffuse;
uniform vec3 L_in;

in vec3 N_v2f;

layout(location = 0) out vec4 C_out;

// Fragment shader entry point
void main() {
  // Compute diffuse lighting
  vec3 N = normalize(N_v2f);
  vec3 L = normalize(L_in);
  C_out = C_diffuse * max(0, dot(N, L));
}
```

Listing 2.3: HLSL: `diffuse.hlsl` *1st part*

```
// Vertex shader entry point
void VertexMain(
  uniform float4x4 wvpMatrix,
  uniform float3x3 wMatrix,
  in      float4   V_in  : POSITION,
  in      float3   N_in  : NORMAL,
  out     float4   V_v2f : SV_Position,
  out     float3   N_v2f : NORMAL)
{
  // Transform coordinate/normal vector
  V_v2f = mul(wvpMatrix, V_in);
  N_v2f = mul(wMatrix,   N_in);
}
```

Listing 2.4: HLSL: `diffuse.hlsl` *2nd part*

```
// Fragment shader entry point
void FragmentMain(
  uniform float4 C_diffuse,
  uniform float3 L_in,
  in      float3 N_v2f : NORMAL,
  out     float4 C_out : SV_Target)
{
  // Compute diffuse lighting
  float3 N = normalize(N_v2f);
  float3 L = normalize(L_in);
  C_out = C_diffuse * max(0, dot(N, L));
}
```

Listing 2.5: Metal: `diffuse.metal`

```
#include <metal_stdlib>

using namespace metal;

struct VertexIn {
  float4 V [[attribute(0)]];
  float3 N [[attribute(1)]];
};

struct VertexOut {
  float4 V [[position]];
  float3 N;
};

// Vertex shader entry point
vertex VertexOut VertexMain( VertexIn           v         [[stage_in]],
                             constant float4x4& wvpMatrix [[buffer(1)]],
                             constant float3x3& wMatrix   [[buffer(2)]] )
```

```
19 {
20
21   // Transform coordinate/normal vector
22   VertexOut v2f;
23   v2f.V = wvpMatrix * v.V;
24   v2f.N = wMatrix   * v.N;
25   return v2f;
26 }
27
28 // Fragment shader entry point
29 fragment float4 FragmentMain( VertexOut        v2f       [[stage_in]],
30                               constant float4& C_diffuse [[buffer(3)]],
31                               constant float3& L_in      [[buffer(4)]] )
32 {
33   // Compute diffuse lighting
34   float3 N = normalize(v2f.N);
35   float3 L = normalize(L_in);
36   return C_diffuse * max(0.0, dot(N, L));
37 }
```

GLSL Example Review

The GLSL implementation of our example points out, that the vertex and fragment shaders must be separated into multiple sources or rather source files (shown in Listings 2.1, 2.2). This also applies to any other shader stage. One of the reasons for this restriction is the `main`-function, the so-called *entry point*, which must have the same function signature[1] in every GLSL shader. The *input* values are either the vertex attributes (`V_in` and `N_in` in `diffuse.vert`), or the data from the previous shader stage (`N_v2f` in `diffuse.frag`). The *output* values are either the data that are going to be passed on to the next shader stage (`N_v2f` in `diffuse.vert`), or the fragment output (`C_out` in `diffuse.frag`). For the obsolete GLSL versions 110 and 120, input and output fields where both specified with the **`varying`** keyword, but we focus on the modern GLSL versions that range from 130 to 460 at the present time. For certain Input/Output (IO) fields there is a reserved word that begins with "`gl_`" such as `gl_Position` and from now on we will call these reserved IO fields *system value semantics*, as they are called that way in HLSL, due to their special meaning for the rendering system.

The rest of the shader is straightforward: **`uniform`** data fields denote constant values that are provided by the host application (i.e. the program running on the Central Processing Unit (CPU)) and they are, as the name implies, uniform for each shader invocation[2]; the data types `vec3/4` and `mat3/4` denote vector and matrix types respectively; the **`layout`**`(location = index)` qualifier[3] specifies the IO slot; and the math functions `normalize`, `max`, and `dot` are predefined functions, so-called *intrinsics*.

HLSL Example Review

HLSL supports multiple entry points in a single source file, which is illustrated by the functions `VertexMain` and `FragmentMain` (shown in Listings 2.3, 2.4). Even though all shader inputs and outputs are declared uniformly inside function parameter lists, HLSL supports several different ways to do that, e.g. by using the function return type, wrapping parameters into structures, or declaring the **`uniform`** data fields outside the function declaration just like in the GLSL example. Unlike GLSL, HLSL provides the `mul` intrinsic rather than an operator for vector and matrix multiplication. Moreover, system value

[1] *Function signatures* define input and output of functions or methods.
[2] *Shader invocation* denotes the execution of a shader for a single thread on the GPU.
[3] *Layout qualifiers* in GLSL affect where the storage for a variable comes from.

semantics are declared with reserved semantic names rather than variable names (here `SV_Position` and `SV_Target`); all other semantics are user-defined[1]. Note that all semantics are *case insensitive*, but for convenience, all user-defined semantics are written in upper-case letters.

The rest of the shader is quite similar to the GLSL example, except that the vector and matrix type names consist of their scalar type and dimensions.

Metal Example Review

In the Metal example we can again define multiple shader entry points within a single source file, just like in HLSL. Additionally, the shader type is specified for each entry point as well (here with the keywords `vertex` and `fragment` in Listing 2.5). Among the different syntax of attributes (e.g. `[[stage_in]]` in the Metal example), the presence of *pointers* and *references* is a major difference between Metal and the previous examples. The uniform values are passed into the shaders with *constant references* (e.g. `constant float4x4 &`). It is one of the side-effects of the closer coupling between Metal and C++14, compared to the rather weak coupling between HLSL/GLSL and C. Another adoption from C++14 is the inclusion of the language's standard library (i.e. `metal_stdlib` for Metal), while in HLSL and GLSL the standard library (which contains all intrinsics for instance) is included in the language itself and does not require or support a manual inclusion. As long as the `metal` namespace is not resolved (like in line 3 of Listing 2.5), all vector/matrix types and intrinsics must be called with its namespace prefix (e.g. `metal::float4`).

The rest of the shader looks quite similar to the HLSL and GLSL examples, so the function bodies in all three examples are somehow related.

2.1.1 Language Feature Comparison

After we have seen an example of a simple graphical shader implementation in three different shading languages, it's time to assemble a feature comparison between these languages. This is important to understand on which key elements the conversion or rather cross-compilation operates. As mentioned earlier, the examples above essentially illustrate the differences of input and output semantics between the languages. We will later discuss additional alternatives to declare these data fields. Beyond IO semantics, there are several other features with different support in GLSL, HLSL, and Metal. The most remarkable feature differences are listed in the following table:

Feature	**GLSL**	**HLSL**	**Metal**
Separation of Textures and Samplers	Only for Vulkan	✓	✓
Implicit Type Conversion	Restricted	Extensive	Extensive
Object-Oriented Intrinsics	Few Exceptions	✓	✓
Multiple Entry Points	✗	✓	✓
Default Arguments	✗	✓	✓
Type Aliasing	✗	✓	✓
L-Values of Input Semantics	✗	✓	✓
Embedded Structures	✗	✓	✓
Member Functions	✗	✓	✓
Inheritance	✗	✓	✓

[1] User-defined semantics are supported since HLSL 4.0; Before, only pre-defined semantics were supported.

When translating HLSL into GLSL for instance, all those features that are not supported in GLSL must be handled and converted properly, and sometimes the avoidance of overhead such as wrapper functions[1] is barely feasible.

2.2 Formal Languages

As a crossover between shaders and compilers we first consider formal languages which are the basis for every programing language. Formal language theory was initiated by Noam Chomsky in the 1950s [16], and provides a minimal mathematical representation to describe language phenomena. Therefore, formal languages are used for the specification of programming languages, which is necessary to develop the respective compiler. Let us begin with a few definitions for the further proceeding:

Definition 1 (adopted from [17]) *An* **alphabet***, denoted as* Σ*, is a finite non-empty set of* symbols*, whereat symbols are meant to be indivisible.*

For example the English language has an alphabet of 52 symbols: the letters 'a'-'z' and 'A'-'Z'; binary code has an alphabet of only two symbols: '0' and '1'; and the alphabet used for many compilers is the American Standard Code for Information Interchange (ASCII) that contains 128 symbols, so-called "characters".

Definition 2 (adopted from [17]) *A* **string** *over an alphabet* Σ *is a finite sequence of symbols of* Σ*, and* ϵ *denotes the empty string.*

For example the well-known hex-code "0xDEADBEEF" is a string from the ASCII alphabet, and "00101010" is a string from both the ASCII and binary alphabets.

Definition 3 (adopted from [17]) *For any alphabet* Σ*, a* **language** *over* Σ *is a set of strings over* Σ*, and its members are also called* words *of the language.*

For example for an alphabet $\Sigma_1 = \{y, e, s, n, o\}$ a valid language could be $L_1 = \{\text{yes}, \text{no}\}$. For a simple language like L_1 it is sufficient to enumerate all words of that language, but for *infinite languages* (i.e. languages with infinite words) another representation is necessary. There are three essential methods to represent such infinite languages: *regular expressions*, *pattern systems*, and *grammars*.

2.2.1 Regular Expressions

Definition 4 (adopted from [17]) *The* **regular expressions** *over an alphabet* Σ *and the languages they represent are defined inductively as follows:*

1. *The symbol* $\varnothing$ *is a regular expression, and represents the empty language.*
2. *The symbol* ϵ *is a regular expression, and represents the language only containing the empty string.*
3. $\forall c \in \Sigma$*, c is a regular expression, and represents the language* $\{c\}$*, whose only member is the string consisting of the single character c.*
4. *If r and s are regular expressions representing the languages R and S, then* $(r|s)$*,* (rs)*, and* (r^*) *are regular expressions that represent the languages* $R \cup S$*, RS, and* R^* *respectively.*

For example $(\texttt{very}\text{␣})^*\texttt{important}$ is a regular expression over $\{a, e, i, m, n, o, p, r, t, v, y, \text{␣}\}$ that matches the words "important", "very important", "very very important", and so on. Note that per definition "very important" for instance is a single word and not two in the sense of formal languages.

Regular expressions, first introduced by [24], are integrated within several programming languages such as ECMAScript, Perl, and Python just to name a few of them. However, the set of all regular expressions cannot be described with a regular expression itself. Therefore, they are inappropriate to describe a complex programming language, or rather a language that is *non-regular*. A common example of such a non-regular language is $\{0^n 1^n | n \geq 1\} = \{01, 0011, 000111, \dots\}$ for which no regular expression exists.

[1] A *wrapper function* is a subroutine that calls primarily a second subroutine.

2.2.2 Pattern Systems

Another method to represent formal languages is to use *pattern systems* [2], which we can define as follows:

Definition 5 (adopted from [17]) *A* **pattern system** *is a triple* (Σ, V, p)*, where* Σ *is the alphabet,* V *is a set of* variables *with* $\Sigma \cap V = \emptyset$*, and* p *is a string over* $\Sigma \cup V$ *called the* pattern*. The language generated by this pattern system consists of all strings over* Σ *that can be obtained from* p *by replacing each variable in* p *with a string over* Σ*.*

For example the pattern system $(\{\mathrm{a}, \mathrm{b}\}, \{v_1\}, v_1 v_1)$ describes the language which consists of all strings that are the concatenation of two equal substrings for $\Sigma = \{\mathrm{a}, \mathrm{b}\}$, namely the set $\{xx | x \in \{\mathrm{a}, \mathrm{b}\}^*\}$ (where "$\underbrace{\mathrm{abbab}}_{v_1}\underbrace{\mathrm{abbab}}_{v_1}$" and "$\underbrace{\mathrm{abba}}_{v_1}\underbrace{\mathrm{abba}}_{v_1}$" are valid words for instance), which is not a regular language, but a so-called *pattern language*.

Although regular expressions and pattern system are quite differently, they are both not expressive enough to be used for complex programming languages.

2.2.3 Grammars

We need a more general system for representing languages, and *formal grammars* are perhaps the most conducive notion for that.

Definition 6 (adopted from [17]) *A* **grammar** *is a quadruple* (Σ, V, S, P)*, where:*

1. Σ *is a finite non-empty set called the* terminal alphabet*, whose elements are called* terminals*.*
2. V *with* $V \cap \Sigma = \emptyset$ *is a finite non-empty set, whose elements are called* non-terminals *or* variables*.*
3. $S \in V$ *is called the* start symbol*.*
4. P *is a finite set of* rules *(also called* productions*) of the form* $\alpha \to \beta$*, where* $\alpha \in (\Sigma \cup V)^* V (\Sigma \cup V)^*$ *and* $\beta \in (\Sigma \cup V)^*$*, i.e. both* α *and* β *are strings of terminals and non-terminals, but only* α *contains at least one non-terminal.*

Let us consider the following example of a very primitive English grammar $G_1 = (\Sigma, V, S, P)$, which is defined as follows:

- $\Sigma = \{\mathrm{I}, \mathrm{greet}, \mathrm{see}, \mathrm{you}, \mathrm{me}, \textvisiblespace, .\}$ is the terminal alphabet (Note that this time Σ comprises *words* rather than *letters*).
- $V = \{\mathit{Sentence}, \mathit{Subject}, \mathit{Verb}, \mathit{Object}\}$ is the set of non-terminals.
- $S = \mathit{Sentence}$ is the start symbol.
- P has the following production rules:

$$
\begin{aligned}
\mathit{Sentence} &\to \mathit{Subject}\textvisiblespace\mathit{Verb}\textvisiblespace\mathit{Object}.\\
\mathit{Subject} &\to \mathrm{I}\\
\mathit{Subject} &\to \mathrm{you}\\
\mathit{Verb} &\to \mathrm{greet}\\
\mathit{Verb} &\to \mathrm{see}\\
\mathit{Object} &\to \mathrm{you}\\
\mathit{Object} &\to \mathrm{me}
\end{aligned}
$$

With the grammar G_1 we can produce English sentences like “I greet you.”, “you see me.” for instance. Now recall the example of the non-regular language $\{0^n1^n | n \geq 1\}$ we have seen earlier: Let be $G_2 = (\{0,1\}, \{S,T,O,I\}, S, P)$, where P has the following production rules:

$$S \rightarrow OT$$
$$S \rightarrow OI$$
$$T \rightarrow SI$$
$$O \rightarrow 0$$
$$I \rightarrow 1$$

It can easily be seen, that the grammar G_2 satisfies the non-regular language from above.

Grammars are classified in a hierarchy of four types of grammars, called the *Chomsky Hierarchy*, due to the work of Noam Chomsky [16]. Each higher layer in the hierarchy decreases the extent of languages that can be represented by a grammar of that layer. This hierarchy is illustrated in Figure 2.3 and the description of each layer is outlined in the following table:

Type	**Name**	**Remarks**
Type-0	Unrestricted/ Recursively enumerable	Languages of Type-0 can be accepted by a Turing Machine (TM)
Type-1	Context-sensitive	Languages of Type-1 can be accepted by a Linear-bounded Automaton (LBA)
Type-2	Context-free	Languages of Type-2 can be accepted by a Push-down Automaton (PDA)
Type-3	Regular	Languages of Type-3 can be accepted by a Deterministic Finite Automaton (DFA)

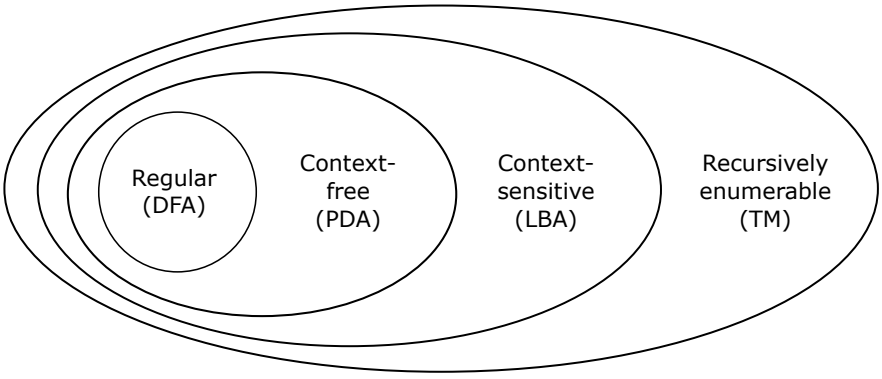


Figure 2.3: Chomsky hierarchy of formal languages/grammars.

The syntax of most programming languages is context-free (or very close to it), and it can be described in a more convenient way, known as the Extended Backus Naur Form (EBNF) [50]. With the EBNF the production rules of the example grammar G_2 from above can be abbreviated as follows:

$$S \rightarrow OT \mid OI$$
$$T \rightarrow SI$$
$$O \rightarrow 0$$
$$I \rightarrow 1$$

The widespread parser[1] generator tool ANTLR [38] also uses a syntax which is similar to the EBNF. We will later see a few examples where ambiguities in the production rules make it hard to implement a fully context-free parser, and some workarounds are necessary.

2.3 Compilers

We are now going over to compiler theory. Like all native programming languages, a compiler is required to translate the high-level code into low-level machine code. A *cross-compiler* (sometimes referred to as *trans-compiler*, *source-to-source compiler*, or *transpiler*) usually translates high-level code from one language into another high-level code that is equivalent in the target language. This process is comparable to the translation of English into German for instance, because both languages are comparatively similar in their extent of expressions. In the following sections we will go through all stages of a very basic and non-optimizing compiler pipeline as shown in Figure 2.4. Many compilers have additional stages, e.g.

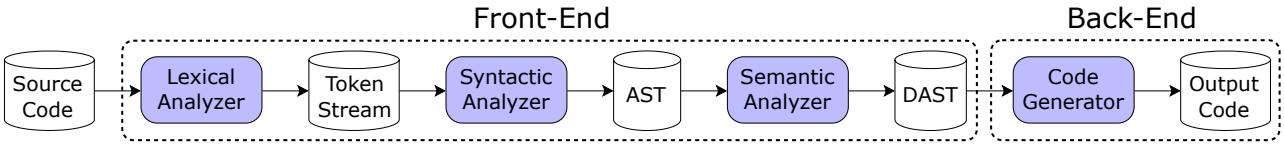


Figure 2.4: Graph of a basic compiler pipeline; White ≙ data models, Blue ≙ compiler stages.

for pre-processing the source code or optimization passes, which will be mentioned shortly. While the *Front-End* is the part which reads the input code and transforms it into something the compiler can work with, namely the Abstract Syntax Tree (AST), the *Back-End* is the part which produces the output code, which can be either high-level code or some sort of assembly (i.e. *binary* or *byte code*).

2.3.1 Lexical Analyzer

The *lexical analyzer*, also called *scanner*, reads the input code and scans all *tokens*. For a lexical analyzer each character of the input code is a *terminal* and tokens are *words* in the sense of formal languages. Commonly a token is a structure that holds at least a string of characters and the type of token classes.

Now recall the example of a very basic English language from earlier: Let the set of token classes be $T = \{Subject, Verb, Object, Space, Dot\}$. Whenever the lexical analyzer reads the character '.' it outputs a token of class T_{Dot}, when it reads the character '␣' it outputs a token of class T_{Space}, when it reads the sequence of characters "see" it outputs a token of class T_{Verb} and value "see", and so on and so forth. We can also define a grammar for a lexical analyzer that accepts this language: Let $\Sigma_{LA} = \{e, g, I, m, o, r, s, t, u, y, ␣, .\}$, $V_{LA} = \{T, Subject, Verb, Object, SubjectObject, Space, Dot\}$, $G_{LA} = \{\Sigma_{LA}, V_{LA}, T, P_{LA}\}$, and the production rules for P_{LA} in EBNF are:

$$
\begin{aligned}
T &\rightarrow Subject \mid Verb \mid Object \mid SubjectObject \mid Space \mid Dot \\
Subject &\rightarrow \text{I} \\
Verb &\rightarrow \text{greet} \mid \text{see} \\
Object &\rightarrow \text{me} \\
SubjectObject &\rightarrow \text{you} \\
Space &\rightarrow ␣ \\
Dot &\rightarrow .
\end{aligned}
$$

[1] *Parsers* are the part of a compiler that read and accept/reject the input code; more about parsers in section 2.3.

Note that we need the variable *SubjectObject* $\in V_{LA}$ for the string “you”, to avoid ambiguities.

In summary, the lexical analyzer abstracts the input code from a stream of single characters into a stream of tokens, which makes it easier for the syntactic analyzer to transform the input code into an AST. For the syntactic analyzer we can then define another grammar where each token is a *terminal* and the AST is a *word* in the sense of formal languages. But before we continue with the syntactic analyzer we first look at the AST, which is a fundamental data structure to represent source code inside a compiler.

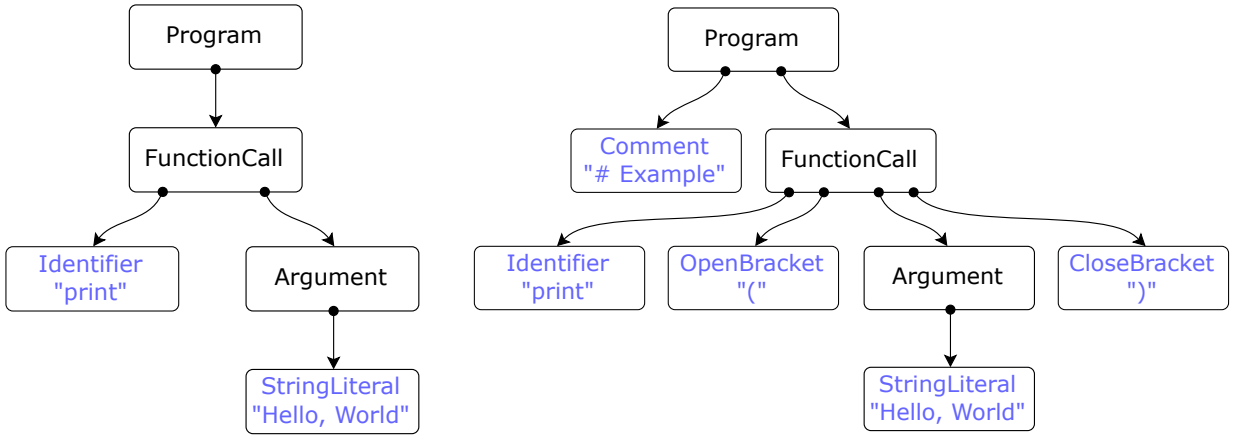
2.3.2 Abstract Syntax Tree

The AST is, as the name implies, an abstract representation of the syntax in a tree hierarchy. What makes the AST *abstract* is that it has all the information removed which is unnecessary for the meaning of the program. These redundancies, such as commentaries and punctuation in programming languages, are called *syntactic sugar* and might only be used to simplify the parsing process and/or assist the programmers to understand the program structure. The counterpart to an AST is a Concrete Syntax Tree (CST) which stores all information even what is declared as syntactic sugar. While in most compilers CSTs are barely used, in a cross-compiler it might be quite reasonable to store all information to provide the same structure and commentaries in the resulting high-level code for instance.

Consider the following example of a very simple program written in *Python*[1] version 3:

```
1 # Example
2 print("Hello, World")
```

This example only consists of a function call with a string literal as argument, and a preceding commentary. A possible variant of an AST and CST for this sample code is illustrated in Figure 2.5. This data



(a) AST **(b)** CST

Figure 2.5: AST and CST examples; Black ≙ non-terminal, Blue ≙ terminal.

structure is much handier for a compiler to work with instead of the source code represented as text, i.e. a series of characters. The major algorithms to traverse and manipulate this tree hierarchy are the *Visitor Pattern* and *Observer Pattern* [49]. A compiler typically uses one of these two design patterns.

[1] *Python* is a scripting language which is (among other things) widely used for introduction to computer programming [44].

2.3.3 Syntactic Analyzer

The *syntactic analyzer*, also called *syntax analyzer* or *parser*, takes the tokens from the lexical analyzer and converts it into an AST. The AST is utilized to store the information of the appliance of the grammar production rules.

As an example, we now define a grammar to construct an AST for the Python example from above: Let $\Sigma_{SA} = \{\text{OpenBracket}, \text{CloseBracket}, \text{Identifier}, \text{StringLiteral}\}$ be the set of tokens, $V_{SA} = \{\mathit{Program}, \mathit{FunctionCall}, \mathit{Argument}\}$ be the AST nodes, $G_{SA} = \{\Sigma_{SA}, V_{SA}, \mathit{Program}, P_{SA}\}$ be the parser grammar, and the production rules for P_{SA} in EBNF are:

$$\begin{aligned} \mathit{Program} &\rightarrow \mathit{FunctionCall}^* \\ \mathit{FunctionCall} &\rightarrow \text{Identifier OpenBracket } \mathit{Argument} \text{ CloseBracket} \\ \mathit{Argument} &\rightarrow \text{Identifier} \mid \text{StringLiteral} \end{aligned}$$

Note that the commentaries and white spaces are simply ignored by the lexical analyzer, i.e. the parser does not even notice these tokens. The asterisk in $\mathit{FunctionCall}^*$ means that the *Program* AST node may consist of zero, one, or more sub nodes of type *FunctionCall*, which allows the program to contain several function calls in sequence.

Parsers are classified into two kinds of algorithms: LL(k) and LR(k) parsers. LL parsers are *top-down* parsers which read the input from Left-to-Right and apply the Leftmost production rule, while LR parsers are *bottom-up* parsers which read the input from Left-to-Right and apply the Rightmost production rule. The k in LL(k) and LR(k) specifies the number of tokens the parser can look ahead, whereat LL(*) specifies a varying amount [40]. The minimum amount of token lookahead is $k = 1$, otherwise the parser could only accept a single word, like a human language that only allows a single sentence.

As mentioned earlier, there are tools for automatic parser generation like ANTLR, but several compilers and compiler frameworks exist where the parser is still written by hand. One of those compiler frameworks is Low Level Virtual Machine (LLVM), which is widely adopted as groundwork for modern compilers [29, 30].

2.3.4 Semantic Analyzer

The *semantic analyzer*, also called *context analyzer*, validates the AST and *decorates* it with references between the nodes. The modified AST is then called a Decorated Abstract Syntax Tree (DAST). While the syntactic analyzer only validates the syntax during the parsing process, the semantic analyzer validates the semantic or rather context information. For example the English sentence *"The circle has angled corners."* is syntactically correct, i.e. the grammar is not violated, but it doesn't make any sense when we take the context into account, that a circle has no corners at all. In the case of a programming language, this could mean that the semantic analyzer rejects a program where a variable is used which has not previously been declared. In the Python example from above the semantic analyzer needs to know the context information that `print` is a function that is always present in a Python program, while other functions must be declared before they can be used. To analyze those identifiers, a *symbol table* is commonly used.

Furthermore, the type system of the programming language is applied in this stage. Type casts both implicit and explicit as well as type compatibility are analyzed according to the type system. Most shading languages have built-in types for vectors and matrices since they are frequently used in shaders. These vectors and matrices are then part of the type system. The complexity of the symbol table and whether multiple symbol tables are required depends on the extent of the type system.

In shading languages there are various attributes and semantics that are only available in a certain shader stage. For example, the depth value of a fragment can obviously be written only in a fragment shader. This must also be analyzed by the semantic analyzer and a proper error report should be emitted in case of failure.

2.3.5 Control Flow Graph

A compiler which provides optimization passes, data structure analysis, or produces assembly code typically has a part between the Front-End and Back-End, called the *Middle-End* (see Figure 2.6). In this

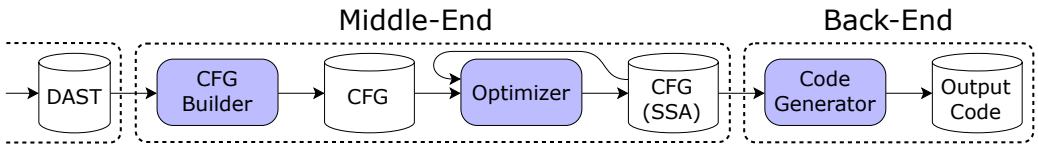
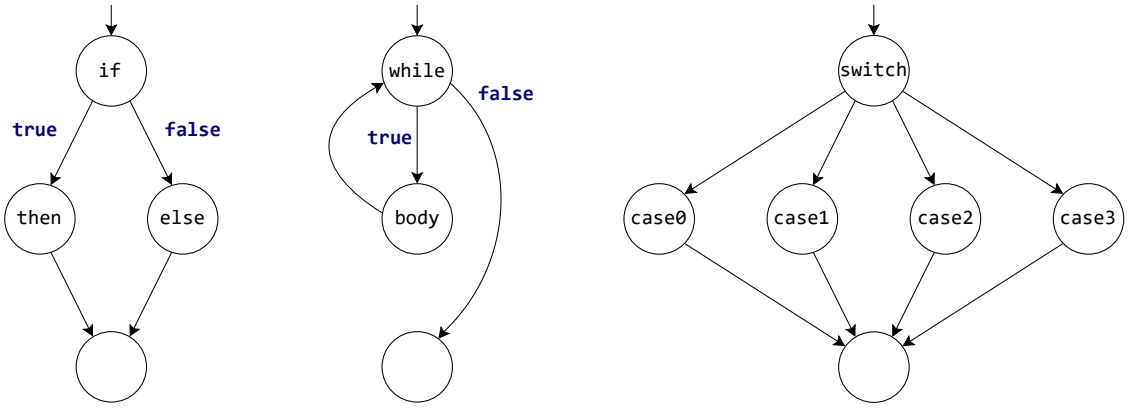


Figure 2.6: Continued graph of an optimizing compiler pipeline, where a *Middle-End* is intervened.

part the compiler operates on the Control Flow Graph (CFG) rather than the AST, which provides a graph structure that represents not only the syntax of the program but also its control flow [1]. Since optimizing compilers and assembly code generators are beyond the scope of this thesis, we only briefly examine the CFG and the compiler stages of the Middle-End.

To convert the DAST into a CFG a dedicated stage in the compiler pipeline is necessary (named "CFG Builder" in Figure 2.6). The nodes of a CFG are called *Basic Blocks* and each of these nodes contains a series of abstract assembler instructions, so-called *op-codes* and sometimes called Three-Address Code (TAC), with an unlimited amount of registers (or rather variables). For example SPIR-V specifies the shader op-codes in the Vulkan API. The edges of the CFG represent "jumps" which the code generator either removes or translates into assembler jump instructions (like `JMP` from the x86-64 instruction set). A few examples of CFGs are illustrated in Figure 2.7. In the early days of hardware accelerated shaders



(a) `if` condition **(b)** `while` loop **(c)** `switch` statement

Figure 2.7: CFG examples, where each node represents a so-called *Basic Block*.

GPUs were not able to perform so-called *conditional jumps*. The shader compilers translated the code so that all *branches* were always being executed but only the desired result was passed to the output. Although this issue has been tackled since the dawn of GPGPU, it still exists in another form. However, the problem has been transfered into the graphics driver.

Especially for optimizing compilers there is a specific form in which the CFG must be for several optimization passes which is called the Static Single Assignment (SSA) form [5]. In this form each variable is assigned only once which makes it easier to implement certain optimization algorithms. At the point where branches are merged, a special instruction, called the "phi" or Φ instruction, is inserted to select the variables from the previous control paths. This Φ instructions must be later resolved when the CFG is converted back from SSA form. For more details about the basics of compiler design the reader is referred to [33].

3 Related Work

In this chapter we will analyze several approaches for shader cross-compilation that are still being used or even at the cutting edge. We will then make a comparison to summarize their benefits and failings.

3.1 Proceedings

3.1.1 Macro Expansion

One of the easiest ways to develop cross platform shaders is the manual approach with *macro expansion*. Macro expansion is a simple language feature implemented by the preprocessor of shader compilers, which is supported by all mainstream shading languages, namely GLSL, HLSL, and MSL. During this process certain tokens in the shader code are replaced by another series of tokens whereat neither type systems nor scopes are applied. This verbatim replacement allows a programmer to make static changes in the code before the actual compilation begins. Not only shading languages have a preprocessor but also general purpose programming languages utilize it such as C and C++. For example the type name of a vector type can be replaced by a name that is supported by the target shading language. The following listings illustrate this idea where each appearance of the token `VECTOR4` is replaced by either `vec4` (for GLSL) or `float4` (for HLSL), and also the type cast is replaced for the respective language:

Listing 3.1: Before preprocessing

```
1 #if defined GLSL
2 #   define VECTOR4    vec4
3 #   define CAST(T, V) T(V)
4 #elif defined HLSL
5 #   define VECTOR4    float4
6 #   define CAST(T, V) (T)(V)
7 #endif
8 VECTOR4 v = CAST(VECTOR4, 0);
```

Listing 3.2: After preprocessing (GLSL)

```
8 vec4 v = vec4(0);
```

Listing 3.3: After preprocessing (HLSL)

```
8 float4 v = (float4)(0);
```

All lines in the code beginning with '#' have a declarative meaning exclusively for the preprocessor. An identifier created with the `define` keyword is called *macro* and only present during preprocessing. When such an identifier appears in the code after it has been defined the preprocessor replaces it with the stream of tokens declared on the right hand side of its definition. If the macro is used in conjunction with arguments (i.e. expressions separated by commas inside parentheses) the macro is expected to have enough parameters which are replaced by these arguments. This replacement process is called *macro expansion*. One of the most prominent shaders making use of macro expansion is the implementation of Fast Approximate Anti-Aliasing (FXAA) [31]. This *post-processor*[1] can be included directly into a shader written in GLSL or HLSL via the heavy use of macros for types, textures, samplers, and other language constructs.

A common coding convention is to write all macros in upper-case letters and separating names with underscores, due to the lack of scopes or rather namespaces. With this coding convention programmers try to avoid overlapping names between macros and other identifiers. It's a necessary evil which is not only cumbersome, but also no guarantee that the preprocessor could eventually replace some identifiers unintentionally. Moreover, detecting errors caused by macro expansion is difficult since the compiler operates solely on the preprocessed code which becomes more incoherent to the input code the more

[1] *Post-processors* are shaders applied to a graphics system which augment or modify the image of the previous render pass.

macros are used. Nevertheless, macro expansion is still used for cross platform shaders, especially for *in-house solutions* like shown in a presentation from Valve during *Steam Dev Days 2014* [8].

3.1.2 Meta-Language & Graph-Based Shaders

Macro expansion can already make shaders look like they were written in an entirely different language, but having a dedicated language with different back-ends is a more elegant solution. A meta-language can be utilized with a design that fits all needs of the target languages. As an example, *TypeScript* [13] provides a stronger type system than its target language *JavaScript*, while at the same time being backwards compatible. It could therefore be called a meta-language.

Such a meta-language requires the development of an entirely new shading language with the respective compiler, so instead of a textual meta-language a common implementation is a *graph-based* shader tool. This visual approach allows the shader designer to create and concatenate nodes that represent small predefined code pieces like multiplication of two input images for instance. No actual programming is necessary which makes it eligible to artists with less programming skills. Similar to the CFG in a compiler, the graph in the shader tool represents the entire program flow of the shader. When the graph is complete the shader tool can generate high-level shader code for the target language. An example of this idea is illustrated in Figure 3.1. Some of those shader tools are also referred to as *graph-based*

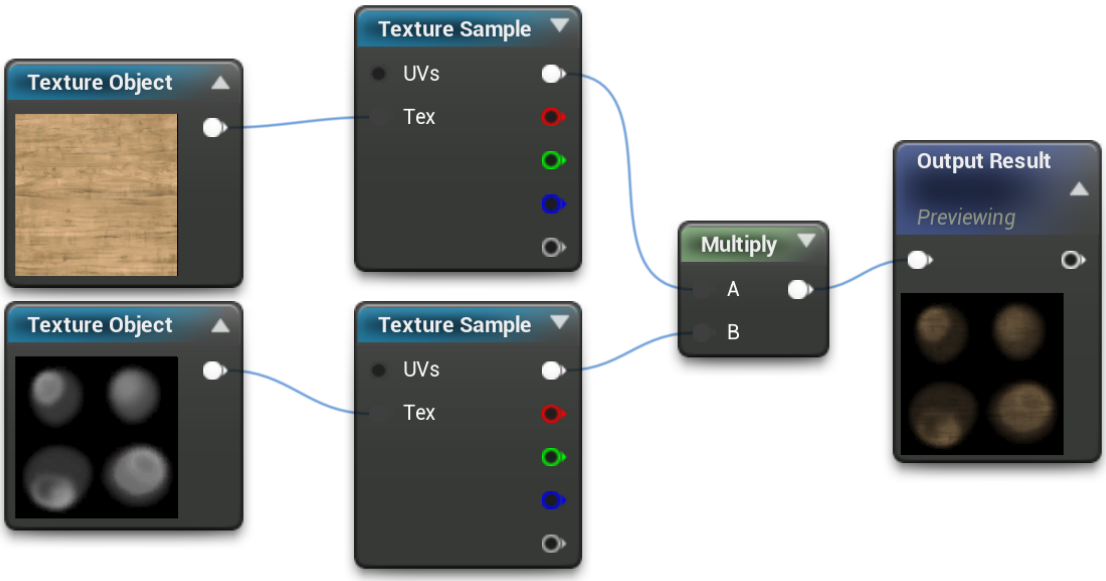


Figure 3.1: Graph-based shader multiplying two textures. Created with “Blueprints” in *Unreal Engine 4* [7].

material systems, and some are more extensive than others. I.e. material systems, as the name implies, commonly only allow developing shaders for material characteristics and might be somehow restricted compared to general purpose shaders.

Though, a graph-based approach can be quite handy, especially because the designer gets an immediate response how the resulting shader will look like. On the other hand, large graphs can quickly become confusing compared to shader source code, especially when several conditions, loops, and other dynamic control flow constructs are included. Another disadvantage is that graph-based shader tools are generally tied to a software framework. The example in Figure 3.1 for instance is part of the *Unreal Engine 4* and cannot be simply integrated into another game engine, not only due to its license. Therefore, using a textual meta-language can be more advantageous when used in various software infrastructures. This in turn requires the developers to learn a new language which may not be very widespread. Hence, a

cross-compiler which supports translation between native shading languages that are well-known might be a better choice.

3.1.3 Byte-Code Decompiling

In the OpenGL render system the shaders are submitted to the GPU driver in their high-level source form, and the graphics driver takes all the compilation work. In contrast, the Direct3D render system takes the shaders in a *non-native binary* form, so-called *byte-code*, which is compiled in advance by a dedicated shader compiler which is separated from the driver. For those render systems for which a dedicated compiler exists that produces byte-code, a cross-compiler could translate this byte-code into high-level source code of another language. This process is commonly known as *decompiling*, which is the reverse of *compiling* high-level to low-level code. The approach has various advantages: the cross-compiler does not need to take care of all the high-level semantics like the type system for instance; code optimizations are also handled by the byte-code compiler; and it's usually much simpler to parse byte-code rather than high-level source code. Furthermore, the translation from low-level to high-level code is basically easier to implement than it is vice versa. For example the byte-code might have a limited amount of registers which can easily be translated to the unlimited amount of variables in the high-level code.

Besides these advantages, there are also various disadvantages with this approach: the byte-code might not be documented, due to a proprietary language as it was the case for a long time with HLSL[1]; the byte-code compiler might not be platform independent which makes the cross compilation somehow restricted; and the generated output code might be unrelated to the input code, which makes debugging more difficult.

Nevertheless, several projects exist following this approach. A few of the more well-known projects are: *HLSLCrossCompiler* by *James Jones* [18], *HLSLcc* by *Mikko Strandborg* [51], *MojoShader* by *Ryan Gordon* [10], and *ToGL* by *Valve Corporation* [4].

3.1.4 Source-to-Source

Next, we will consider *direct* source-to-source translation, i.e. the translation or rather transformation operates on the AST only and no CFG or a certain IR is generated. This approach requires at least a parser for the source language and a code generator for the destination language. All the other stages of a full-fledged compiler are more or less optional. For some programming languages it is sufficient to only parse the source code and output new high-level source code. In this case, however, (semantically-) invalid input code leads to invalid output code instead of a sophisticated error report. Recall the small code sample from Listing 3.1 where we only transform the vector type for either GLSL or HLSL. This can also easily be solved without macro expansion when a parser is available, because the types can be trivially mapped to another name. For the differences of IO semantics between the shading languages, significantly more transformations need to be done on the AST. Especially the transformation of structural differences requires a high workload, e.g. object-oriented concepts compared to data-oriented concepts.

It would be optimal if existing shader code could be translated without any modifications. However, in some cases a 1:1 mapping between two shading languages is not possible without additional information. One example is the description of data formats: in GLSL the data format of image buffers is specified in detail with a layout qualifier, e.g. `layout(rgba32f)` specifies an RGBA image format with a 32-bit single precision floating-point data type for each color component. On the other hand, HLSL leaves the detail of bit-size to the host application, e.g. `float4` can be used for an RGBA image format with floating-point data but of unspecified bit-size. Hence, translating HLSL to GLSL requires additional information for some buffer objects. A crucial advantage of a custom source-to-source compiler is that the source

[1] In 2017 the new HLSL compiler, which is based on Clang/LLVM, has been released as open-source by Microsoft.

language can be extended to circumvent these limitations. We will later discuss how those language extensions can be implemented while keeping the divergence of the source language low.

As long as the cross-compiler is meant to only support one single source language, the data structures for the AST can be automatically generated by a parser generator tool like ANTLR. Otherwise, the auto-generated AST structures might be inappropriate to be used for several source languages, due to a different tree hierarchy. Hence, a manually written parser including its AST structures are beneficial for a multi-directional cross-compiler. To better understand why this is the case, take a look at the following example of a so-called *constant buffer*[1] in GLSL (Listing 3.4) and HLSL (Listing 3.5).

Listing 3.4: GLSL Constant Buffer

```
1 layout(binding = 0) uniform Example {
2   /* ... */
3 };
```

Listing 3.5: HLSL Constant Buffer

```
1 cbuffer Example : register(b0) {
2   /* ... */
3 };
```

If we used a parser generator, we would need two grammar specifications, illustrated in Listings 3.6 and 3.7, where non-terminals are written in lower-case and terminals are written in upper-case. For more details about the ANTLR syntax, the interested reader is referred to [39].

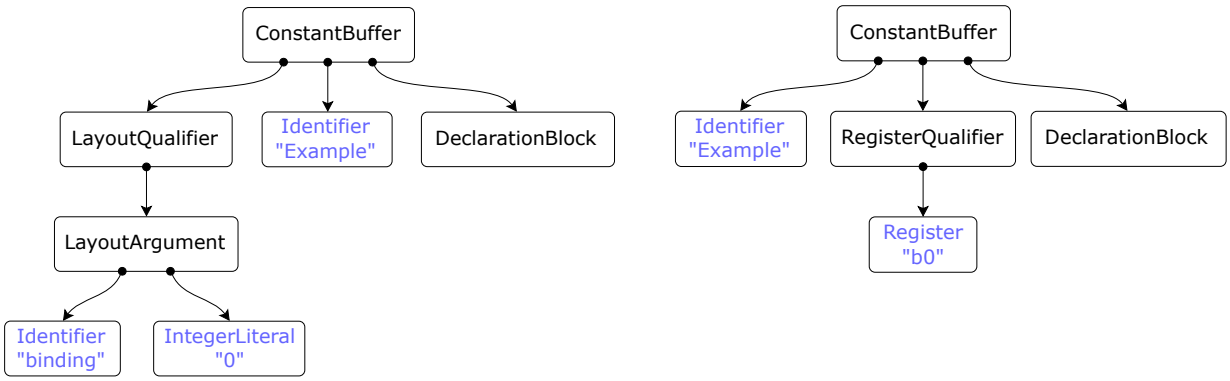
Listing 3.6: ANTLR grammar for GLSL

```
1  constant_buffer:
2    layout_qualifier?
3    'uniform' IDENTIFIER
4    declaration_block;
5
6  layout_qualifier:
7    'layout' '(' layout_argument ')';
8
9  layout_argument:
10   IDENTIFIER '=' INTEGER_LITERAL;
```

Listing 3.7: ANTLR grammar for HLSL

```
1  constant_buffer:
2    'cbuffer' IDENTIFIER
3    (':' register_qualifier)?
4    declaration_block;
5
6  register_qualifier:
7    'register' '(' REGISTER ')';
8
9  REGISTER:
10   'b' [0-9]+;
```

With these two grammars we would end up with two different sets of AST structures like the two shown in Figure 3.2. If the AST structures are written manually the parser can generate the tree hierarchy uni-



(a) GLSL **(b)** HLSL

Figure 3.2: Different sets of AST structures, one for each language; Black ≙ non-terminal, Blue ≙ terminal.

formly for each source language. This also implies that the AST structures must have a common denominator, which inevitably leads to a little overhead, but this is negligible. For example the `ConstantBuffer` AST node could always have a sub node for both `LayoutQualifier` and `RegisterQualifier` and only use the one which is needed, depending on the source language. As already said, those AST unifications

[1] A *constant buffer* provides constant input data for the shader which is uploaded from CPU to GPU.

are only necessary (or rather meaningful) if the compiler supports multiple source languages. Otherwise, the compiler also gets along with automated techniques. In most cases, the latter is sufficient, but we will come back to this subject later.

Besides the AST and parser construction, another important scope of work in a source-to-source compiler for shading languages is the mapping between intrinsics. Many intrinsics can be adopted directly, such as the `normalize` intrinsic which is identical throughout GLSL, HLSL, and MSL. Other intrinsics can be mapped just by their names, such as `bitCount` (in GLSL) to `countbits` (in HLSL). And still others must be transformed in a more specific manner. In the next chapter we will discuss the concept of such transformations in more depth.

Various projects exist following the approach of source-to-source translation. One of the projects that merely has a parser and code generator is *HLSLParser* by *Unknown Worlds* [55], which translates HLSL to GLSL. Unfortunately, it only supports the obsolete HLSL Shader Model (SM) 3 for Direct3D 9.0c, which was introduced in 2004. The latest version of HLSL in 2017, however, is SM 6.0 for Direct3D 12. Another project that does a little more than just parsing the input code is *HLSL2GLSLfork* by *Aras Pranckevičius* [41]. This cross-compiler uses, among other things, a symbol table for contextual analysis. However, it only supports HLSL SM 3 as well or rather Cg which is quite similar. And not to forget *ANGLE* by *Google* [9], which is used in the Chrome browser to map the shaders (but also the rendering commands) between WebGL and Direct3D.

For the sake of completeness, the ROSE compiler infrastructure [42] is mentioned here, which is mainly used for source-to-source transformation and analysis for large-scale programs written in C, C++, Fortran, and other CPU languages. This framework is not directly related to shading languages but could also be used for such a cross-compiler, since shading languages are based largely on C. However, it has not been adopted in shader cross-compilers so far.

3.1.5 Intermediate Language

Last but not least, we will examine the cross-compilation with an IR or rather an intermediate language. In this proceeding a full-fledged compiler is required, i.e. all stages of a compiler pipeline are involved, including the middle-end which generates the CFG and IR instructions. In contrast to compilers for CPU languages, a shader compiler does not generate native assembly code like it is the case with the x86 architecture (also known as IA-32). In other words, the graphics driver translates the assembly code or IR once more into another native assembly code. Modern shader compilers don't even have to go deeper into the matter than the IR, which is commonly some sort of *high-level assembler*. The major reason for this is that the native assembler language of GPUs is vendor specific and generally inaccessible. As a result, the hardware vendors can constantly change their GPU architecture without invalidating any shaders. Furthermore, the low-level optimizations that are most suitable for the respective hardware device are done by the driver.

Before the appearance of high-level shading languages, the shaders were written manually in a non-native assembly language. They were introduced in OpenGL 1.5 and Direct3D 8.0, and were called "vertex/fragment programs" or simply "asm shaders" [12]. Another assembly language for shaders is Adobe Graphics Assembly Language (AGAL) which was developed for *Adobe Flash* [53]. Since it is only meant to be used for the Web-based Flash player we don't go into more details about AGAL. These assembly languages were very low level since even *register allocation* must be taken into account, i.e. there is only a very limited set of variables available. In contrast, an IR usually supports unlimited variables. So we will distinguish these two kinds of low level shading languages as assembly languages and IRs. Most assembly shading languages are obsolete today, so we will focus on IR shading languages from now on.

The first open standard of an IR specifically for shading languages is SPIR 1.2 for OpenCL, which is based on LLVM IR and released to the public in 2013 [37]. It has later been redesigned by the *Khronos Group* as a stand-alone specification without LLVM but with graphics support, which is named SPIR-V

[19]. Since *Microsoft* made its HLSL compiler open-source in 2017, DirectX Intermediate Language (DXIL) is the second open standard of an IR for shading languages which is also based on LLVM IR [32].

SPIR-V

A single shader unit in the SPIR-V binary form is called a *module*. Besides code validation, glslang, the official reference compiler for GLSL, can produce such modules as output. As mentioned in the introduction, the Khronos Group has published various tools to generate high-level shader code out of those modules. The major tool for this proceeding is SPIRV-Cross [3]. SPIR-V provides adequate information of types and names to make cross compilation easy. Otherwise, the resulting high-level code would primarily contain indexed identifiers which makes the code hard to understand.

A special characteristic of SPIR-V modules is that they are always in SSA form, as defined in the specification [21]. This allows the graphics driver to perform vendor specific optimizations immediately. To get an overview of how a SPIR-V module can look like, see the illustration of a simple GLSL fragment shader in Listing 3.8 and the resulting module in Listing 3.9:

Listing 3.8: GLSL Fragment Shader

```
1 #version 450
2
3 // Declare fragment output attribute
4 layout(location = 0) out vec4 color;
5
6 void main() {
7     // Set white output color
8     color = vec4(1.0);
9 }
```

Listing 3.9: SPIR-V Module

```
1         OpCapability Shader
2   %1 = OpExtInstImport "GLSL.std.450"
3         OpMemoryModel Logical GLSL450
4         OpEntryPoint Fragment %4 "main" %9
5         OpExecutionMode %4 OriginUpperLeft
6         OpSource GLSL 450
7         OpName %4 "main"
8         OpName %9 "color"
9         OpDecorate %9 Location 0
10  %2 = OpTypeVoid
11  %3 = OpTypeFunction %2
12  %6 = OpTypeFloat 32
13  %7 = OpTypeVector %6 4
14  %8 = OpTypePointer Output %7
15  %9 = OpVariable %8 Output
16 %10 = OpConstant %6 1
17 %11 = OpConstantComposite %7 %10 %10 %10 %10
18  %4 = OpFunction %2 None %3
19  %5 = OpLabel
20        OpStore %9 %11
21        OpReturn
22        OpFunctionEnd
```

The fragment shader has been compiled with *glslang* and the module has been disassembled with *spirv-dis*. Among other things, the example illustrates how the IR stores debug information. For example the `OpName` instruction in line 8 specifies the name of the output variable `color`. We can also see immediately that the module is in SSA form because each index (`%1` to `%11`) is assigned only once. A cross compiler, like SPIRV-Cross, can then consume the SPIR-V module and output high-level code. The reconstruction of data types and control flow is straightforward. However, it is somewhat more complex to produce an output that is as close as possible to the original code. In other words, sometimes wrapper functions and several temporary variables are inevitable.

Both glslang and SPIRV-Cross are still under development, especially the support of HLSL (as of 2017). Nevertheless, they form a solid foundation for cross-compilation in a multi-directional and modular manner. Especially the official specification of an IR for shading languages is an important step forward to tackle former issues like divergent compiler behavior between different drivers. Moreover, SPIRV-Cross supports all modern shading languages for real-time graphics: GLSL, HLSL, and MSL. Even C++ code can be produced by SPIRV-Cross for offline shader debugging. An outline of the operation mode between glslang and SPIRV-Cross is shown in Figure 3.3. Another SPIR-V based cross-compiler is *krafix* by *Robert Konrad* [26]. A special feature of *krafix* is that it provides a transformation from OpenGL specific GLSL

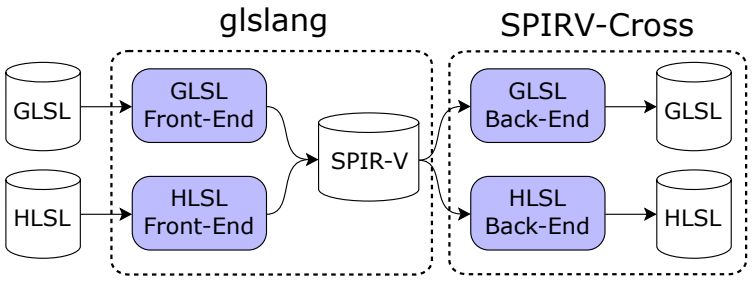


Figure 3.3: Operation mode between glslang and SPIRV-Cross; illustrated for GLSL and HLSL.

into Vulkan specific GLSL, since there are a few subtle differences between the GLSL specifications for OpenGL and Vulkan. This transformation only operates on SPIR-V code.

DXIL

DXIL can be seen as the counterpart to SPIR-V which is used for the new DirectX Shader Compiler (DXC) and still under development (as of 2017). The DXC was not meant to be used as cross-compiler originally. In conjunction with SPIR-V the development of cross-compilation has recently begun, though. Although a direct translation from DXIL to SPIR-V would be theoretically possible, the responsible developers have chosen a different design: Front-end AST to SPIR-V. One of the reasons for this design decision is that DXIL has a lower level of abstraction than SPIR-V. For example vector types are converted into scalars but SPIR-V has native support for vectors. Another reason is that DXIL and SPIR-V have very different semantics. Among other things, structured control flow is required by SPIR-V but DXIL does not support it. Since the new DXC is based on the Clang C++ compiler, the semantic analysis step is appropriate to generate error reports when cross-compilation fails. And at this point of the compiler pipeline, Clang still operates on the AST. The DXC also distinguishes between DXIL and DirectX Intermediate Representation (DXIR) which has a higher level of abstraction than DXIL. However, since DXIR is not suitable for fast Just-in-Time (JIT) compilation, the optimizer transforms DXIR into DXIL.

There is an open discussion and documentation for the translation of semantic differences where other developers can contribute to. As already mentioned, sometimes a 1:1 translation is not feasible without additional information, so-called *meta-data*. One example for a proposal of such meta-data is shown in Listing 3.10.

Listing 3.10: HLSL with meta-data for SPIR-V

```
1 [[using Vulkan: set(X), binding(Y)]]
2 cbuffer Buffer {
3     [[using Vulkan: offset(Z)]]
4     float4 field;
5 };
```

The additional information for the Vulkan rendering system in this example is: the set of buffers, the binding slot, and memory offset. The syntax for these meta-data (specified by `[[...]]`) has been adopted from the C++14 *attribute specifiers*.

3.2 Comparison

After we have seen various approaches for shader cross-compilation we can make a comparison between the proceedings by highlighting their benefits and failings. The most advanced proceeding is clearly the use of an IR together with a complete compiler pipeline. This approach scales well to support multiple

source and target languages, provided that the CFG generation out of the AST is generalized. However, this approach naturally requires the highest effort. A summarized outline of all examined proceedings can be seen in the following table:

Pros	Cons
Macro Expansion	
• Very easy to implement • No tool chain required	• Inappropriate to port existing shaders • Hard to debug and maintain • Tied to software framework • Inappropriate as general purpose solution • Confusing for large shaders
Meta-Language	
• Language for requirements of all targets • Potentially easier to add further back-ends	• Inappropriate to port existing shaders • Requires new/elaborated specification • New language specification rather unknown • High effort
Graph-Based Shaders	
• Language for requirements of all targets • Potentially easier to add further back-ends • Visual approach good for artists	• Inappropriate to port existing shaders • Tied to software framework • Confusing for large shaders • High effort
Byte-Code Decompiling	
• Appropriate to port existing shaders • High-level compilation not required. • Translation is much easier.	• Only feasible for certain shading languages • Portability issue of high-level compiler • Output code unrelated to source code • Undocumented/closed byte-code format
Source-to-Source	
• Appropriate to port existing shaders • Allows extensions for interoperability • Output code notably related to source code	• No validation vs. full-fledged compiler • Appropriate semantic translation is hard • High effort
Intermediate Language	
• Appropriate to port existing shaders • Allows extensions for interoperability • Easier to add further back-ends	• Requires full-fledged compiler with CFG • Very high effort

4 Concept

As we have seen in the previous chapter, shader cross-compilation is still a topical issue. The support for multiple source and target languages in state-of-the-art shader compilers is still in development (as of 2017). And the infrastructure of various modern shader compilers have partially divergent code bases, like glslang which uses different internal data structures for GLSL and HLSL. We will now investigate the concepts for a novel shader cross-compiler with the main focus on GLSL and HLSL, since they are the most widely used shading languages for real-time graphics.

4.1 Approach

In this thesis we will adopt the approach of a **source-to-source** compiler instead of using an IR. This is a reasonable compromise between the amount of effort and the output quality. Especially the relation between the source code and the target code can be maintained very well. This makes the compiler much more beneficial for translation of existing shaders where the output code might be maintained and extended manually. In the following sections we will examine the major issues and differences between GLSL and HLSL like the storage layouts, IO semantics, type conversion, intrinsics, and others. The conceptual compiler we design here is also meant to have one uniform AST which will be discussed in more depth in chapter 5.

4.2 Investigation

Which kind of parser a cross-compilers uses and how the data structures look like are part of the implementation details. The most important part for the conceptual translation, however, is the semantic difference. We will now see how these translations can be done in principle and where compromises have to be made. No matter how good a shader cross-compiler is, some features are only available in a certain language. We use "GLSL→HLSL" for "GLSL to HLSL translation" and "HLSL→GLSL" as opposite.

4.2.1 Matrix Ordering

A GPU almost always operates on 4-component vectors. Matrix operations are therefore split into subsequent vector operations. So either the rows of the matrix or the columns of the matrix must be moved into registers. Which of these two methods is used determines the *matrix ordering*, which is *column-major* by default and can be changed to *row-major* for each matrix in both GLSL and HLSL. However, the definition of matrix dimensions differs: in GLSL it is **column-by-row**, and in HLSL it is **row-by-column**. As a result, for a matrix $A_{ij} \in \mathbb{R}^{4\times4}$, A_{12} for instance refers to the matrix element at column 1 and row 2 in GLSL, but it refers to the matrix element at row 1 and column 2 in HLSL. And it becomes even more problematic when non-squared matrices are used.

One solution to resolve this difference is to swap the row- and column dimensions and modify each element access: $A_{ij} \in \mathbb{R}^{N\times M}$ becomes $A_{ji} \in \mathbb{R}^{M\times N}$, and each element access A_{ij} becomes A_{ji}. Although this solution seems to be quite easy, it requires a lot of effort because shading languages allow matrix elements to be accessed in many ways. Such a transformation is illustrated in Listing 4.1 and Listing 4.2.

Listing 4.1: Matrix Dimension Swap in GLSL

```
1 mat4x2 A;
2 vec2 a1 = vec2(A[2][1], A[2][0]);
3 vec2 a2 = vec2(A[2].y, A[2].x);
4 vec2 a3 = A[2].yx;
```

Listing 4.2: Matrix Dimension Swap in HLSL

```
1 float2x4 A;
2 float2 a1 = float2(A[1][2], A[0][2]);
3 float2 a1 = float2(A[1][2], A[0][2]);
4 float2 a3 = float2(A[1][2], A[0][2]);
```

In line 3 and 4 it can be seen that the translation from GLSL to HLSL requires significantly more effort than just swapping the dimensions.

Another and more elegant solution is to keep the matrix dimensions as they are and only change the matrix ordering and swap the arguments in every matrix multiplication which is equivalent to transpose the matrix. Let $v \in \mathbb{R}^M$ be an M-dimensional vector and $A \in \mathbb{R}^{N \times M}$ be an $N \times M$-dimensional matrix. We can now perform the following mathematical transformation:

$$A \times v = (A \times v)^{TT} = \left(v^T \times A^T\right)^T \tag{4.1}$$

Here, $()^T$ denotes the transpose matrix or transpose vector respectively. With this transformation we can keep the matrix dimensions and also every matrix element access. Moreover, the only transpose which needs to be computed is the transpose matrix, because in shading languages there is no distinction between 'row vectors' and 'column vectors'. Therefore, the resulting transformation is $A \times v \rightarrow v \times A^T$. Whether a matrix is transposed or not depends on the order of the arguments within a matrix-vector multiplication. This is done with the *-operator in GLSL and with the `mul` intrinsic in HLSL. An exemplified translation is shown in Listing 4.3 and Listing 4.4.

Listing 4.3: Matrix Ordering in GLSL

```
1  // A has 4 columns and 2 rows
2  layout(row_major) uniform mat4x2 A;
3                    uniform vec2   v;
4
5  // Vector-Matrix multiplication
6  vec2 w = v * A;
7
8  // Matrix element access
9  vec2 a1 = vec2(A[2][1], A[2][0]);
10 vec2 a2 = vec2(A[2].y, A[2].x);
11 vec2 a3 = A[2].yx;
```

Listing 4.4: Matrix Ordering in HLSL

```
1  // A has 4 rows and 2 columns
2  column_major uniform float4x2 A;
3               uniform float2   v;
4
5  // Matrix-Vector multiplication
6  float2 w = mul(A, v);
7
8  // Matrix element access
9  float2 a1 = float2(A[2][1], A[2][0]);
10 float2 a2 = float2(A[2].y, A[2].x);
11 float2 a3 = A[2].yx;
```

The matrix ordering has been changed from *row-major* to *column-major*, which changes the order in which the rendering system reads the matrix elements from memory. This ensures that the graphics programmer does not need to change the CPU code whether GLSL or HLSL is used, although the matrix dimensions have been swapped. In GLSL, `v*transpose(A)` is equivalent to `A*v` and the same holds true for HLSL with the `mul` intrinsic. In this way, each matrix element access can stay as it is but the shader code still keeps the same behavior. This potentially minimizes the transformations and the output code is almost synchronous with the input code.

4.2.2 Matrix Subscript Swizzling

The components of a vector can be accessed by a *vector subscript*. It is written as suffix after a vector expression, e.g. `myVector.x` to access only the X-axis of the variable `myVector`. The *swizzle operator* (also referred to as *swizzling*) is a feature in shading languages to rearrange the order of vector components in a handy way, e.g. from a 4D-vector v the components can be accessed via `v.xyzw`, `v.zxzx`, or `v.xxxw` to form new vectors. This feature allows to access multiple vector components at once and also multiple times.

HLSL supports the subscript also for matrices, but GLSL does not. The translation of accessing single matrix elements is not a problem, but when it comes to *swizzling* it gets more complicated. HLSL provides two syntaxes for the matrix subscript: a *zero-based* index (e.g. `_m01` for the first row and second column), and a *one-based* index (e.g. `_12` for the same element). These indices can be combined in the same way as for vectors. An example of a matrix subscript using the swizzle operator, with A being a 4×4 matrix, is illustrated in Listing 4.5 and Listing 4.6.

Listing 4.5: Matrix subscript in HLSL

```
1 A._m00_m00_m00_m33
```

Listing 4.6: Matrix subscript in GLSL

```
1 vec4(A[0][0], A[0][0], A[0][0], A[3][3])
```

Both code samples reflect the same 4D-vector. The translation looks quite simple but it is invalid when the matrix expression has side effects. For example, instead of `A.` the expression could be a function call like `A().`, with `A` being a function that returns a matrix. The compiler must not generate multiple calls of that function, otherwise the program behavior may change. This is the case when the function also writes something into an output buffer and/or increments a global counter for instance. A simple solution would be to generate a wrapper function for each index combination that is used in the shader. This wrapper function takes only one parameter which can then be duplicated as desired without any side effects. The expression inside the return statement of this wrapper function looks then like the one in Listing 4.6.

This works great for read-access but write-access is another issue. The write-access could be solved by a second version of this wrapper function but this alone is not sufficient when output parameters are involved. In case of output parameters, the insertion of additional statements is unavoidable. An illustration is provided in Listing 4.7 and Listing 4.8.

Listing 4.7: Matrix subscript in HLSL

```
1 void f(inout float2 v);
2 /* ... */
3 f(A._m00_m11);
```

Listing 4.8: Matrix subscript in GLSL

```
1 void f(inout vec2 v);
2 /* ... */
3 vec2 temp = read_m00_m11(A);
4 f(temp);
5 write_m00_m11(A, temp);
```

In this example, `f` is a function within the shader that takes a 2D-vector as input and output parameter, and the wrapper functions `read_m00_m11` and `write_m00_m11` were generated by the compiler. Unfortunately, this approach for matrix subscript translation might generate relatively lot of auto.-generated code, but is owed by the lack of a language feature in the target language. Instead of generating different wrapper functions for zero-based and one-based indices, it is sufficient to agree on a single variant. The insertion of additional statements can be tricky to implement, because a matrix subscript expression can appear almost everywhere even where temporary variables are not allowed, such as in the iteration expression of a `for`-loop. More details about the implementation details are discussed in chapter 5.

4.2.3 Memory Packing Alignment

Buffer objects in shading languages have a *memory packing alignment*, also referred to as *byte alignment*. This might introduce additional fields in the buffer, so-called *padding*. How many padding bytes are inserted depends on the type of the data fields. The common rule is that a data field must not cross the boundaries of a 4-component vector. Each component of such a vector has a size of 4 bytes. In this way, the data can be read more efficiently. Sometimes padding is inevitable, but usually it can be reduced at least. Consider the following data structure for a light source which is illustrated as pseudocode in Listing 4.9 (with padding) and Listing 4.10 (without padding):

Listing 4.9: *Pseudocode*: Structure with padding

```
1 struct LightSource:
2     float1 alpha  // x
3     float4 color  // x y z w
4     float3 pos    // x y z
5     float3 dir    // x y z
6     float1 radius //       w
```

Listing 4.10: *Pseudocode*: Structure w/o padding

```
1 struct LightSource:
2     float4 color  // x y z w
3     float3 pos    // x y z
4     float1 alpha  //       w
5     float3 dir    // x y z
6     float1 radius //       w
```

The letters `x`, `y`, `z`, and `w` illustrate to which vector components the data fields are assigned. Although the order `wxyz` seems to be more sensible, `xyzw` is the common order of vector components in the

nomenclature of computer graphics. Both versions of the `LightSource` structure have the same data fields but in different order. How the vectors are laid out in registers can be seen in Table 4.1 (with padding) and Table 4.2 (without padding). Because a 4D vector like `color` is always stored inside a

Register v_0				Register v_1				Register v_2				Register v_3			
x	y	z	w	x	y	z	w	x	y	z	w	x	y	z	w
alpha	*padding*	*padding*	*padding*	color	color	color	color	pos	pos	pos	*padding*	dir	dir	dir	radius

Table 4.1: Registers for structure with padding

Register v_0				Register v_1				Register v_2			
x	y	z	w	x	y	z	w	x	y	z	w
color	color	color	color	pos	pos	pos	alpha	dir	dir	dir	radius

Table 4.2: Registers for structure without padding

single register, it can be loaded as fast as possible. As a result, the packing alignment for buffer objects in shaders is 4 Bytes × 4 Components = 16 Bytes. Reducing the amount of padding or even completely avoiding it has only optimization purposes. But it is important to keep this technique in mind when buffer objects are created, because the structure layout must be synchronous between CPU and GPU.

Now that we know what the memory packing alignment and padding in shaders is, we can cross over to its translation between GLSL and HLSL. In HLSL it always works as described above, but in GLSL it only works equally when it is explicitly specified. To enable the well-defined packing alignment behavior in GLSL, a buffer must be declared with the `std140` or `std430` layout qualifier (depending on the buffer type). Otherwise, the driver is free to handle the memory packing differently in which case the memory location of each buffer element must be queried by the host application. Because only the CPU-side is affected whether the respective buffer layout qualifier is used or not, this translation is very straightforward. The compiler could offer the programmer to either enable or disable the generation/removal of the respective layout qualifier. In the simplest way, the GLSL→HLSL translation could always generate the layout qualifier per default. A small example is illustrated in Listing 4.11 and Listing 4.12.

Listing 4.11: Constant Buffer in GLSL

```
1 layout(std140, binding = 0)
2 uniform Example {
3     vec4  color;
4     vec3  pos;
5     float alpha;
6     vec3  dir;
7     float radius;
8 };
```

Listing 4.12: Constant Buffer in HLSL

```
1 cbuffer Example : register(b0) {
2     float4 color;
3     float3 pos;
4     float  alpha;
5     float3 dir;
6     float  radius;
7 };
```

By using `std140` for the constant buffer (also referred to as Uniform Buffer Object (UBO) in GLSL), the data can be uploaded to the GPU driver equally between GLSL and HLSL. For a Read/Write (RW) buffer (also referred to as Shader Storage Buffer Object (SSBO) in GLSL) it would be `std430` in the layout qualifier.

4.2.4 Type Conversion

Just like general purpose programming languages, shaders support type conversion, too. A typical example is the conversion from integral types to IEEE[1] floating-point types. As already mentioned in the table of section 2.1.1, GLSL has a quite restricted support for implicit type conversion compared to HLSL. In other words, GLSL requires almost always an explicit type conversion that is declared with the respective syntax. Even the conversion between signed and unsigned integral types must be specified explicitly (i.e. `int` to `uint` and vice versa). This also applies to integral literals: for example `uint i = 0;` is an invalid assignment in GLSL, but `uint i = 0u;` is valid because the literal `0u` is explicitly specified as unsigned integral type with the 'u' suffix. HLSL on the other hand does a lot more of this conversion automatically. For this reason, the GLSL→HLSL translation is very simple in this context, because the extensive use of explicit type conversion is optional in HLSL. However, the HLSL→GLSL translation is much more complicated. In fact, a complete type system within the compiler is required to determine the type of each expression. The syntactic command for a type conversion is called a *type cast*, which we have already seen in Listing 3.1.

What we need to do in our conceptual compiler to translate implicit- into explicit type conversion, is to insert a type cast into each expression whenever two types cannot be implicitly converted within the target language. How this insertion is done will be discussed in chapter 5. First we need to determine *where* such type conversions are necessary. For this analysis a sophisticated type system is indispensable.

Type System

The type system of our compiler consists of a set of several type classes. Each expression has a unique type which is assigned during the *semantic analysis*. In a general purpose programming language like C++ the type system is very complex, because a type can be modified multiple times by *templates*, *references*, and *pointers* for instance. Since MSL is based on C++14, this also applies to this language. However, since we are only focusing on GLSL and HLSL, we can be satisfied with a simpler type system. A proposal for a set of type classes is shown in the following table:

Type	**Remarks**
VOID	Type class for functions without return type.
NULL	Placeholder for legacy HLSL syntax when a texture object is initialized with `NULL`.
BASE	Type class for base data types: *Scalar*, *Vector*, *Matrix*.
BUFFER	Type class for buffer objects: *Texture Buffer* and *Storage Buffer*.
SAMPLER	Type class for *Samplers* and *Sampler States*.
STRUCT	Container with multiple fields which in turn also have a type.
ALIAS	Type alias which refers to another type.
ARRAY	Type class for arrays with a certain dimension and a sub type.

These type classes are sufficient for cross-compilation between GLSL and HLSL, whereat the type ALIAS is only required for HLSL→GLSL, due to the lack of type aliasing in GLSL. Scalars, vectors, and matrices can be grouped into the type class BASE, but other approaches are also possible. However, a lower number of type classes can reduce the amount of effort significantly.

As soon as an expression is traversed during the semantic analysis, the type of this expression is deduced and stored inside the AST node. The AST nodes for the following expression classes are notably important for the type deduction: *Binary Operator* (e.g. `X+Y`), *Ternary Operator* (e.g. `X?Y:Z`), and *Object Identifier*. The binary operator for example might have different deduction rules depending on the

[1] *IEEE 754* specifies a standardization for binary representation of floating-point values for both CPUs and GPUs.

associativity of the respective operator. And for the object identifier the corresponding type of the identified object must be deduced. If the type of an expression could not be deduced, a sophisticated error report should be emitted. In a compiler which generates an IR the type system must also store low-level information such as *bit-size*. But in a source-to-source compiler the type system does not need that information as long as the types behave in the same way within all supported languages. Not only whether explicit or implicit type conversion is supported but also *how* a type is deduced depends on the source language. The following table shows a rough mapping between source and target types where an explicit conversion is necessary in GLSL:

Source Type	Boolean	¬Boolean	Signed Integral	Unsigned Integral	Integral	Real
Target Type	¬Boolean	Boolean	Unsigned Integral	Signed Integral	Real	Integral

For a full classification the interested reader is referred to the GLSL specification [20].

Built-in Variables

We have already seen a few *system value semantics* earlier like `SV_Position`. In GLSL they are implemented as so-called *built-in variables* like `gl_Position`. These built-in variables are available throughout the entire shader, while the semantics in HLSL are assigned to local variables. This is essentially a syntactic difference, except that a couple of built-in variables in GLSL, which have the same meaning as their corresponding semantic in HLSL, have slightly different types. One example is the zero-based vertex index which is defined as `int gl_VertexID` in GLSL and as `uint SV_VertexID` in HLSL. As mentioned above, these types (i.e. signed and unsigned integrals) must be explicitly converted in GLSL.

4.2.5 Input and Output

The translation of IO data fields is probably one of the most important and most complex parts of shader cross-compilation. While various shaders don't necessarily need any type conversion, almost every shader makes use of the language specific IO. Both GLSL and HLSL provide multiple ways to specify the input and output fields between shader stages, but they are fundamentally different. While GLSL uses a syntax where the IO fields are globally declared, just like the built-in variables, HLSL provides a syntax for the *semantics* of local variables. These differences are also caused by the fact that GLSL does not allow parameters or return values for the main function. In HLSL, on the other hand, IO is provided by parameters and return values exclusively. An example of various IO fields is illustrated in Listing 4.13 and Listing 4.14.

Listing 4.13: Fragment shader IO in GLSL

```
1  in Input {
2    vec4 ambient;
3  } inData;
4
5  in vec4 diffuse;
6
7  out vec4 color;
8
9  void main() {
10   color = inData.ambient + diffuse;
11 }
```

Listing 4.14: Fragment shader IO in HLSL

```
1  struct Input {
2    float4 ambient : AMBIENT;
3  };
4
5  void main(
6    in  Input  inData,
7    in  float4 diffuse : DIFFUSE,
8    out float4 color   : SV_Target)
9  {
10   color = inData.ambient + diffuse;
11 }
```

In the GLSL example (Listing 4.13) we have a so-called *interface block* named `Input`, which is accessed by its *scope name* `inData`. A similar construct is used in HLSL (Listing 4.14) with a structure named `Input`, which is accessed by the parameter named `inData` in the main function. Depending on the direction in

which we translate, we are facing different challenges. For example, if the globally defined fields are used outside the main function, the resulting HLSL code must pass these variables to the respective function calls as arguments. As a result, the signatures of various functions must be transformed. The other way round some identifiers may have to be renamed so there are no name overlaps, which is called *name mangling*. For example, a function with a local variable or parameter named `color` cannot access a global input field with the same name simultaneously. Hence, moving the input parameter `color:SV_Target` from the parameter list into global scope cannot always be done without name mangling.

From GLSL to HLSL

In GLSL there are two ways to specify shader IO: single variables and interface blocks. The interface blocks can be used to group several variables together similar to a structure declaration. They are primarily used to simplify the handling of multiple variables between shader stages, especially for the *geometry-* and *tessellation* shader stages where they are often used as aggregates or rather arrays. All these IO variables that appear in a function other than the main function must be passed around. This means that the corresponding parameter and argument lists must be extended in the output code.

Built-in variables must be converted into local variables with the respective system value semantic, e.g. `in vec4 gl_Position` becomes something like `in float4 gl_Position : SV_Position`. Those single built-in variables can be trivially translated but for aggregates used in a geometry shader for instance the corresponding translation is significantly more difficult. The translation of shader IO in a small geometry shader is illustrated in Listing 4.15 for GLSL and in Listing 4.16 for HLSL.

Listing 4.15: Geometry shader IO in GLSL

```
1 layout(points) in;
2 layout(points, max_vertices = 1) out;
3
4 in vec4 vertexPos[];
5
6 void main() {
7   gl_Position = vertexPos[0];
8   EmitVertex();
9 }
```

Listing 4.16: Geometry shader IO in HLSL

```
1  struct v2g
2    { float4 vertexPos : VERTEXPOS; };
3  struct g2f
4    { float4 gl_Position : SV_Position; };
5
6  [maxvertexcount(1)]
7  void main(in     point       v2g  i[1],
8            inout PointStream<g2f> stream)
9  {
10   g2f o;
11   o.gl_Position = i[0].vertexPos;
12   stream.Append(o);
13 }
```

Such a geometry shader is called a “pass-through” shader which merely passes the input arguments to the output stream. The *aggregate* is declared as `vertexPos[]` in the GLSL code. The colored boxes highlight which information belongs together: GLSL uses the `layout`-qualifier again for both input and output primitive topology[1]. In GLSL the number of input vertices are implicitly determined by the input primitive topology, i.e. 1 for *points*, 2 for *lines*, and 3 for *triangles*. In HLSL it must be explicitly declared within the input parameter, like `point v2g i[1]` in the above example. Also the output stream must be declared as a local variable or rather function parameter (here `stream`) in HLSL. In GLSL, on the other hand, the output stream is handled as global and hidden object which can be accessed via certain intrinsics such as `EmitVertex`. The number of output vertices, however, can be easily translated from the layout attribute `max_vertices = 1` to the standalone attribute `[maxvertexcount(1)]`. In this example it can be seen that the translated code (here Listing 4.16) must introduce new structures that where not provided by the input code. Temporary variables (here `g2f o`) are also necessary, especially when multiple IO variables are used. For simplicity the variable names `i` for “input” and `o` for “output” were used, but a practical compiler should use certain prefixes to avoid overlapping names. The variable name

[1] *Primitive topology* denotes the geometric primitive which are primarily variants of *points*, *lines*, and *triangles*.

`gl_Position` in the structure `g2f` can remain because it's not a reserved word in HLSL. Last but not least the semantic names can be deduced from the variable names and written out in upper case letters for convenience.

Let us now summarize the key points we need to consider when translating shader IO fields from GLSL to HLSL:

- Interfaces blocks can be converted to structures: e.g. `in Input` to `struct Input`.
- Certain layout attributes can be trivially converted: e.g. `max_vertices` to `maxvertexcount`.
- Certain layout attributes serve as type information: e.g. `points` to generic `PointStream` type.
- Certain intrinsics must be concatenated to new objects: e.g. `EmitVertex` to `stream.Append`.
- Certain array indices must be moved to other variables: e.g. `vertexPos[0]` to `i[0].vertexPos`.
- Sometimes auto-generated variables and structures are necessary: e.g. `i`, `v2g`, `stream`, etc.

From HLSL to GLSL

We now examine the opposite direction for shader IO translation. The ways in which IO fields can be specified in HLSL are somewhat more extensive than in GLSL. While in GLSL output fields are always specified as global variables with the `out`-specifier, in HLSL they can be specified as function return value or output parameter. An example of various output fields in HLSL is illustrated in Listing 4.17 and a proposal for a corresponding translation in GLSL is illustrated in Listing 4.18.

Listing 4.17: Vertex shader IO in HLSL

```
1  struct Output {
2    float field1 : FIELD1;
3  };
4  float main(
5    out Output o,
6    out float  field2 : FIELD2) : FIELD3
7  {
8    o.field1 = 1;
9    field2   = 2;
10   return     3;
11 }
```

Listing 4.18: Vertex shader IO in GLSL

```
1  out float FIELD1;
2  out float FIELD2;
3  out float FIELD3;
4
5  void main() {
6    FIELD1 = 1;
7    FIELD2 = 2;
8    FIELD3 = 3;
9  }
```

What should be the first thing to notice is that this time not the variable names were used in the translation but their semantic names (e.g. `FIELD1` instead of `field1`). This is not absolutely necessary but it is advisable for a couple of reasons: Firstly, all semantic names must be unique throughout the entire shader, and second, the names between the shader stages are conventionally almost always identical. The former facilitates the translation to global variables in GLSL. The latter one is not a requirement but almost every shader is written with identical semantic names between stages, while the variable names can often vary, e.g. `position:POS` in the vertex shader but `pos:POS` in the fragment shader. In GLSL, on the other hand, they must necessarily be identical. This is an elegant and straightforward solution, but hazardous as soon as a structure with IO fields is used for additional purposes. Consider that the structure from Listing 4.17 appears in another function as output parameter such as `void Transform(out Output o)`. In this case it is easier to produce the main function in the output code as a wrapper function. The original main function is then translated into a secondary function with a slightly different name which is called inside the wrapper. With this approach the main entry point can almost remain as it is and the IO fields are assigned to its function call. An example is illustrated in Listing 4.19 with the function `_main` as the original entry point. The IO fields are assigned within the actual main function that serves as a wrapper. This is a solution which is frequently used by various shader cross-compilers. Most of them even generate this wrapper function every time. However, as long as shader IO related structures are only used within the main function, those wrapper functions can be avoided. Especially mobile devices have only

limited support for code optimization which requires additional code transformations for optimal executions. Hence, the avoidance of such wrapper functions is desired, but sometimes difficult to implement.

Listing 4.19: Wrapper function in GLSL

```
/* ... */

// Original main function
float _main(out Output o, out float field2) {
  o.field1 = 1;
  field2   = 2;
  return     3;
}

// Main function serves as 'wrapper' to call "_main"
void main() {
  // Assign shader IO fields
  Output o;
  float field2;
  float field3 = _main(o, field2);
  FIELD1 = o.field1;
  FIELD2 = field2;
  FIELD3 = field3;
}
```

Let us again summarize the key points we need to consider when translating shader IO fields, but this time from HLSL to GLSL:

- Semantic names should be preferred over variable names: e.g. `field1:FIELD1` to `FIELD1`.
- Sometimes a wrapper function for the entry point is inevitable *(but not always)*.

The key points for the GLSL→HLSL translation can be adopted analogously, too.

4.2.6 Intrinsics

In GLSL, all intrinsics have the same characteristics as global functions that are known by the compiler throughout the entire shader. The difference is that they don't need to be declared because they are functions which are built into the language itself. The same applies to HLSL except that there are also Object-Oriented (O-O) intrinsics that can be called exclusively with an instance of their respective class. But before we dedicate ourselves to O-O intrinsics, we first consider the global intrinsics.

Procedural Intrinsics

The approach of global intrinsics originates from the *procedural programming paradigm*, when so-called *sub-routines* were arranged into procedures but without further scopes like classes. Before SM 4, there were only global intrinsics even in HLSL. As already explained, various global intrinsics can be translated straightforward by simply renaming them, due to their similarity between GLSL and HLSL, assuming that the parameters are identical. All basic trigonometric functions, for example, do not even need to be renamed and can be adopted without modification. One exception here is the `sincos` intrinsic in HLSL which combines the sine and cosine into a single function call. For this and a few more intrinsics it is reasonable to generate wrapper functions in the output code. Note that several versions of the same wrapper function may be generated under these circumstances, due to the lack of templates or generics. Every time a new type is used with an intrinsic, for which a wrapper is generated, a new version for this wrapper must be generated. Another solution would be to put the modification directly on the spot,

but this will potentially lead to an output code being less related to the source code. Which solution fits best depends on how much code must be added/removed and if output parameters are involved. Output parameters make a direct insertion more difficult especially in conjunction with the swizzle operator. An intrinsic where a direct translation without any wrappers is easily feasible is the `saturate` intrinsic in HLSL which corresponds to the following mathematical definition:

$$saturate: \begin{array}{l} \mathbb{R} \to [0,1] \\ x \mapsto \max\{0, \min\{x, 1\}\} \end{array} \tag{4.2}$$

For example, a function call to the intrinsic `saturate(x)` with floating-point value x can be translated to `clamp(x, 0.0, 1.0)` in GLSL. If the input parameter x is a 4D-vector instead, it could be translated to `clamp(x, vec4(0.0), vec4(1.0))`.

There are also very few intrinsics which just cannot be translated. One of them is the `abort` intrinsic in HLSL which terminates the current draw call completely, i.e. the execution of all shader invocations of the current GPU command. Such an intrinsic simply does not exist in GLSL. Fortunately, those intrinsics are rarely used in productive shaders since they are intended primarily for debugging rather than actual program behavior.

O-O Intrinsics

With the introduction of classes in HLSL 4, O-O intrinsics were also integrated. In GLSL there is only one O-O intrinsic which is `length` to determine the dimensions of arrays at compile time. Not to be confused with the global intrinsic to determine the length of a vector that has the same name. HLSL has classes for all kinds of buffer objects such as *textures* and *structured buffers*. Translating the intrinsics of these classes from HLSL to GLSL can be accomplished in three steps:

1. Translate the intrinsic into the corresponding GLSL intrinsic.
2. Move the expression that specifies the object (or rather class instance) into the argument list.
3. Either join or remove the sampler state object if present.

The other direction works analogously. An example with a couple of intrinsics for texture sampling[1] is illustrated in the following table:

GLSL	**HLSL**
`texture(t, uv)`	`t.Sample(s, uv)`
`textureLod(t, uv, l)`	`t.SampleLevel(s, uv, l)`
`textureGradOffset(t, uv, dx, dy, o)`	`t.SampleGrad(s, uv, dx, dy, o)`
`textureQueryLod(t, uv).x`	`t.CalculateLevelOfDetailUnclamped(uv)`
`textureQueryLod(t, uv).y`	`t.CalculateLevelOfDetail(uv)`

The following variables were used: **`t`** specifies a texture object, **`s`** specifies a sampler state, **`uv`** specifies texture coordinates, **`dx`** and **`dy`** specify derivation vectors in x and y direction respectively, **`l`** specifies a Level Of Detail (LOD), and **`o`** specifies an offset. In GLSL, the separation of textures and sampler states is only available for Vulkan. The texture and sampler state objects are therefore only joined for the Vulkan-specific GLSL. For OpenGL, sampler states simply do not exist in the shading language, although there are sampler objects in the rendering system. As a result, sampler states must be removed for the OpenGL-specific GLSL. This is one of the most significant restrictions in GLSL compared to HLSL. But OpenGL programmers must be aware of these circumstances anyway. Joining a texture with a sampler

[1] *Texture sampling* is a method to read a texture buffer with a certain filter function.

in GLSL is accomplished with an intrinsic to create a new sampler, e.g. `sampler2D(t, s)`. Applying this to the first GLSL example from the table above results in `texture(sampler2D(t, s), uv)`. When we go the other direction, i.e. GLSL→HLSL, we could simply generate a sampler state for each texture. The last two examples with the `textureQueryLod` intrinsic illustrate a special case: Two functionalities have been implemented into a single intrinsic in GLSL, but in HLSL there are two different intrinsics for them. If both vector components are used at once, a wrapper function should be used again.

Let us briefly return to the `length` intrinsic in GLSL: its translation is possible via the type system. The compiler needs to now the dimension of all arrays and also arrays of arrays. For each expression where the intrinsic is used the type system must be able to deduce the array size. In this way the compiler can replace the intrinsic call by a constant integral expression.

4.2.7 Attributes

Attributes in HLSL and layout attributes in GLSL are fundamentally different concepts. In some situations, however, they are intended for the same purpose. In HLSL, attributes can appear almost everywhere, but various attributes are meant to be used only for the main entry point, such as the `maxvertexcount` attribute that we have seen earlier. In GLSL, on the other hand, attributes can only appear inside a layout qualifier which is primarily available for global declarations. A few information declared with a layout attribute in GLSL must be declared with a specific keyword in HLSL. One of them is the **`register`** keyword to specify the binding slot of a constant buffer for instance. In contrast, GLSL uses the `location` attribute.

HLSL Attributes

There are many attributes in HLSL for fine tuning the control flow. They are only used for optimization purposes which are more or less obsolete nowadays. Doing low-level optimizations at high-level source code does not make much sense anymore, since every graphics driver has different ways to perform efficient optimizations for certain architectures. This is why SPIR-V provides the code in SSA form to let the graphics driver do the remaining optimization passes. A brief overview of a couple of those attributes is provided in the following table:

Attribute	**Remarks**
`branch`	Forces the shader to perform dynamic branching for an `if`-statement.
`flatten`	Avoids dynamic branching and evaluates both sides of an `if`-statement.
`unroll`	Removes dynamic control flow of a loop statement by unrolling each iteration.
`call`	Avoids inlining a function call.

These attributes are a relict of the older versions of HLSL. They can appear even in front of a variable declaration where they have no effect. It is reasonable to simply ignore these attributes in a shader cross-compiler, since they do not affect the behavior of the shader. In OpenGL, a shader can either fully enable or disable the optimizer. To disable the optimizer for instance, the preprocessor directive `#pragma optimize (off)` can be used after the GLSL version is specified. But for no specific control flow construct it can be specified how it is meant to be assembled.

The only HLSL attributes we are interested in, are the one to specify global shader properties, like `[numthreads(x,y,z)]` to specify the number of local threads for a compute shader. Again, these attributes can only appear in front of the main entry point. The corresponding layout qualifiers in GLSL, on the other hand, can appear anywhere in the global scope. An example of two attributes for a compute shader and a fragment shader respectively is presented in the following table, where *X*, *Y*, and *Z* specify constant integral expressions:

GLSL	HLSL
`layout(early_fragment_tests) in;`	`[earlydepthstencil]`
`layout(local_size_x=X, local_size_y=Y, local_size_z=Z) in;`	`[numthreads(X,Y,Z)]`

As one can see, those attributes can be trivially translated. However, it is somewhat more complicated with the tessellation control shader or rather "Hull Shader" in HLSL. Recall that tessellation is a graphics feature provided by the hardware which is split up into two shader stages: the *tessellation control-* and *tessellation evaluation* shader. There are several attributes in both GLSL and HLSL to configure the behavior of the hardware tessellator[1]. In HLSL, almost all attributes for the hardware tessellator are specified for the tess.-control shader, while in GLSL, on the other hand, the attributes are differently distributed over the tess.-control and tess.-evaluation shaders. This leads to two problems: Firstly, when compiling the tess.-evaluation shader, the name of the tess.-control shader must also be specified explicitly, and secondly, it may happen that both shaders are not always available at the same time. Knowing the main entry points of both tessellation shaders is necessary to correctly translate the attributes for the target language. When translating GLSL→HLSL we also have the problem that always only one shader can be present in a single compilation unit or source file. Therefore, the cross-compiler must provide a mechanism to specify where the opposite tessellation shader is present. All the attributes related to tessellation shaders (***TC*** for tess.-control; ***TE*** for tess.-evaluation) are shown in the following table with a couple of example arguments:

GLSL	HLSL
N/A	***TC*** `[domain("quad")]`
TC `layout(vertices=4) in;`	***TC*** `[outputcontrolpoints(4)]`
N/A	***TC*** `[patchconstantfunc("MyFunc")]`
TE `layout(fractional_odd_spacing) in;`	***TC*** `[partitioning("fractional_odd")]`
TE `layout(ccw) in;`	***TC*** `[outputtopology("triangle_ccw")]`
TE `layout(quads) in;`	***TE*** `[domain("quad")]`

The attributes that are not available (marked with '*N/A*' in the table) are either implicitly present in GLSL by the vector dimension of a certain IO field, or specified directly inside the main entry point, i.e. there is no need for a function like "`MyFunc`".

GLSL Attributes

In addition to the HLSL attributes there are also some layout attributes in GLSL which must be translated to a register location. They can be used for single uniforms but also for binding slots of textures and samplers. The major attributes for these purposes are either `location` or `binding`. Depending on the type for which the binding point is used, a certain register must be used in HLSL. With a straightforward translation the index slots could simply be adopted. We have already seen this kind of translation in a previous section for a UBO in Listing 4.11 and Listing 4.12. It is important to note here that Direct3D allows resources to be bound to different binding slots for each individual shader stage. For example, a texture can be bound to slot N for the vertex shader, while another texture is also bound to slot N, but for the fragment shader. In OpenGL, on the other hand, each texture is always bound to a global binding slot that affects all shader stages at the same time. Therefore, the naive approach may lead to incorrect resource binding if the shader was not created with this restriction in mind. A possible solution

[1] The *hardware tessellator* is a fixed-function stage in a graphics pipeline which generates many geometric primitives out of a few control points, so-called *patches*

for this issue is to automatically generate binding slots in the GLSL output, i.e. each buffer has its own unique binding slot. The software library of the shader cross-compiler should then provide an interface to query all generated binding slots, so the programmer can use them in the graphics system. All supported registers are listed in the following table with the associated HLSL type and the corresponding GLSL type for both OpenGL and Vulkan:

Register	HLSL Type	GLSL Type for OpenGL	GLSL Type for Vulkan
`b`	Constant Buffer	Uniform Buffer	
`u`	Unordered Access Buffer	Shader Storage Buffer	
`t`	Texture	Sampler	Texture
`s`	Sampler State	*N/A*	Sampler
`c`	Pack Offset	*N/A*	

Note that OpenGL does not support sampler-states in GLSL, but Vulkan does. Samplers are used for texture sampling as a whole in GLSL by default, while the GLSL variant for Vulkan uses samplers for sampler-states and textures for the texture-buffers. Also note that GLSL does not support a *pack offset* which allows the packing alignment of all constant buffer fields to be specified manually. However, the use of the `packoffset` keyword in HLSL is quite restricted. Nevertheless, a cross-compiler should either ignore that keyword or emit a warning report whenever the manually specified packing alignment does not match the default alignment.

4.2.8 Buffer Objects

We are now investigating the mappings between all the classes of buffer objects in GLSL and HLSL. The easiest way would be to always utilize the most flexible class of buffer objects, e.g. a buffer that can always be written to and read from. However, these types of buffers require a lot of GPU power, so they should only be used when they are absolutely necessary. The different buffer types also have varying hardware requirements and are supported by different shader versions. Therefore, a sophisticated cross-compiler should try to find the minimal buffer class for the output code. This allows, among other things, that the output code can also be used on older hardware devices, as far as this is possible. A proposal for a mapping between all buffer types is presented in Table 4.3. As one can see, the type `buffer` in GLSL requires a very high version, namely 430 (460 is the highest version available at the present time). Hence, using this type instead of `samplerBuffer` and `imageBuffer` for instance would be a waste of GPU resources.

Mapping the types alone is only half the work. Using various buffer types implies that different techniques of data access to these buffers must be translated, too. The translation between all the `sampler*`/`Texture*` types is relatively simple, because they behave very similar. There are intrinsics for sampling texture data or reading individual texture elements. Besides the samplers there are also a couple of buffer types with read/write support as well as buffers with generic structured types which are more complicated.

The maximum amount of memory supported for the buffer types is generally the same between GLSL and HLSL. These limitations are primarily hardware dependent, though. The specification of Direct3D 11 guarantees that hardware buffers can have at least 128 MB of memory. Unfortunately, the `glGet*`-functions to query the maximal supported buffer size in OpenGL only provide a rough approximation. Instead, so-called *proxy textures* must be created to determine the actual hardware limitations. Nevertheless, the latest OpenGL and Direct3D versions commonly share the same limitations as long as they are running on the same hardware. Therefore, we can largely ignore the buffer sizes during the cross-compilation. If necessary, we can still offer the user an option to switch to another buffer alternative.

GLSL		HLSL		Dim.	RW	Filter
Type/ Block	SM	Class	SM			
sampler1D	110	Texture1D	4.0	1D	✗	✓
sampler1DArray	130	Texture1DArray	4.0	2D		
sampler2D	110	Texture2D	4.0	2D		
sampler2DRect	140			2D		
sampler2DArray	130	Texture2DArray	4.0	3D		
sampler3D	130	Texture3D	4.0	3D		
samplerCube	130	TextureCube	4.0	2D		
samplerCubeArray	400	TextureCubeArray	4.0	3D		
sampler2DMS	150	Texture2DMS	4.0	2D		✗
sampler2DMSArray	150	Texture2DMSArray	4.0	3D		
samplerBuffer	140	Buffer	4.0	1D		
		ByteAddressBuffer	4.0	1D		
buffer	430	StructuredBuffer	4.0	1D		
imageBuffer	420	RWBuffer	5.0	1D	✓	
		RWByteAddressBuffer	5.0	1D		
buffer	430	RWStructuredBuffer	5.0	1D		
		AppendStructuredBuffer	5.0	1D		
		ConsumeStructuredBuffer	5.0	1D		
image1D	420	RWTexture1D	5.0	1D		
image1DArray	420	RWTexture1DArray	5.0	2D		
image2D	420	RWTexture2D	5.0	2D		
image2DArray	420	RWTexture2DArray	5.0	3D		
image3D	420	RWTexture3D	5.0	3D		

Table 4.3: Buffer type mapping between GLSL and HLSL.

Multiple Mappings

A few types are mapped to multiple opposite types. One of them is `Texture2D` which is mapped to `sampler2D` and `sampler2DRect` where the latter one does *not* use normalized texture coordinates[1]. But in HLSL, texture coordinates are always normalized for sampling. Hence, translating `sampler2DRect` to `Texture2D` requires us to generate a transformation of the texture coordinates including an intrinsic to query the size of the respective texture. Another special case here is `samplerBuffer` that maps to both `Buffer` and `ByteAddressBuffer`. As the name implies, a `ByteAddressBuffer` in HLSL provides data access by a byte address instead of a per-element index. Such a feature is not supported in GLSL so we need to transform the element indexing in our translation. The same applies for the RW-versions of these buffer types. The last type we mention here is **`buffer`** which is mapped to four different kinds of structured buffers. Whether the GLSL type provides write access or not depends on its layout declaration. The last two structured buffer types, namely `AppendStructuredBuffer` and `ConsumeStructuredBuffer`, provide a couple of certain intrinsics to increment an internal counter within the buffer object. With this counter, elements can be appended or consumed. This can only be achieved by a separated object in

[1] *Texture coordinates* are usually normalized to the interval $[0, 1]$ to make them independent of the texture resolutions.

GLSL, a so-called *Atomic Counter*. Using this new counter object of course changes the way the graphics system can interact with the shader IO. HLSL only provides atomic counters as a hidden object within the RW-structured-buffers. Hence, the GLSL→HLSL translation can be more difficult than vice versa, whenever an atomic counter is used in GLSL, especially when it is used for multiple purposes.

Structured Buffers

In HLSL, the structured buffer types have a generic argument to a structure declaration (like the templates in C++). The non-structured buffer types also have a generic argument but only to a base type like `float4`. In GLSL, the corresponding type is an SSBO which is declared with the **`buffer`** keyword. This buffer type does not have a generic 'template-like' argument. Instead, the body of the buffer declaration is written like a structure declaration. As a result, it can have several fields, while a structured buffer in HLSL is always handled like an array of multiple elements of the same structure. To get the same kind of buffer where the size is defined at runtime, GLSL allows the last field to be declared as an array of undefined size. This simplifies the HLSL→GLSL translation, where the buffer can be declared with only a single element which is an array of such an undefined size. This is illustrated in Listing 4.20 and Listing 4.21.

Listing 4.20: Structured Buffer in HLSL

```
1 struct Light {
2   float3 pos, color;
3 };
4 StructuredBuffer<Light> lights;
```

Listing 4.21: Structured Buffer in GLSL

```
1 struct Light {
2   vec3 pos, color;
3 };
4 layout(std430) buffer LightBuffer {
5   Light lights[];
6 };
```

In both cases the object `lights` can be accessed like an array and the number of elements is defined by the graphics system on CPU-side. For the opposite direction, i.e. the GLSL→HLSL translation, the translation only works in the same way if the buffer only has this single dynamic array element. Otherwise, there are two cases: Firstly, none of the elements in the buffer is a dynamic array in which case the buffer can be turned into a new structure declaration and the resulting structured buffer will only have a single element. Secondly, the buffer has multiple elements and one of them is a dynamic array in which case the buffer must be translated into *two* separate buffers in HLSL. The latter one is quite problematic because it involves an entirely new buffer object. This definitely changes the way a programmer can interact with the shader from within the graphics system. An example of such a worst case scenario is illustrated in Listing 4.22 and Listing 4.23.

Listing 4.22: Structured Buffer in GLSL

```
1 struct Light {
2   vec3 pos, color;
3 };
4 layout(std430) buffer LightBuffer {
5   uint lightCount;
6   Light lights[];
7 };
```

Listing 4.23: Structured Buffer in HLSL

```
1 struct Light {
2   float3 pos, color;
3 };
4 struct LightCount {
5   uint lightCount;
6 };
7 StructuredBuffer<LightCount> lightCount;
8 StructuredBuffer<Light> lights;
```

In this case, GLSL is more flexible than HLSL which leads to a restriction for a GLSL→HLSL translation. For very small read-only buffers like the single element buffer `lightCount` in Listing 4.23 it is of course reasonable to use another buffer type such as a *constant buffer*. Unfortunately, the data still needs to be split, which is disadvantageous. This is another compromise that cross-platform shader programmers should be aware of. In this case, it also does not make any difference if we base our cross-compiler on an IR. The restriction remains the same.

RW Buffers

Buffers with access for reading and/or writing are non-trivial to translate. This is because HLSL provides the same syntax for read and write access like for arrays, while GLSL on the other hand has certain intrinsics for this purpose. Which intrinsics are required also depends on the buffer type, because there are different intrinsics for *sampler-buffers* and *image-buffers*. We illustrate this with a simple HLSL `RWBuffer` in Listing 4.24 which is translated to a GLSL `imageBuffer` in Listing 4.25.

Listing 4.24: RW Buffer in HLSL

```
1 RWBuffer<float4> vectors;
2 /* ... */
3 vectors[i] *= 1.5;
```

Listing 4.25: RW Buffer in GLSL

```
1 uniform imageBuffer vectors;
2 /* ... */
3 imageStore(vectors, i, imageLoad(vectors, i) * 1.5);
```

In this example we can see that HLSL allows a direct modification of the elements inside `vectors` (here with the *multiply-assign* operator `*=`). GLSL, however, does not allow a direct modification or rather reading and writing simultaneously. Instead, we need to load and store the data explicitly. The interested reader may have noticed that we have the same problem of duplicated expressions as we had earlier. In this case the expression `i` is duplicated into both image load and store intrinsics. For such an expression with no side-effects it is not a problem to duplicate them. However, if we have a complex arithmetic expression or a function call, duplication would be either inefficient or even results in a different behavior of the translated shader compared to the original shader. In such a case temporary variables are again inevitable and must be carefully inserted. Also note that in HLSL, a vector type is specified for the buffer, but not in GLSL. The intrinsics `imageStore` and `imageLoad` always work with a 4-component vector. To use *signed* or *unsigned* integral vectors, the buffer type must have the 'i' or 'u' prefix respectively. This also applies to all other image and sampler types in GLSL. In other words, the image buffer type is specified as `gimageBuffer` where *g* can be either *i*, *u*, or omitted.

When we use the SSBO type (i.e. the **`buffer`** keyword) in GLSL, we can work with the buffer identically like before. But as already mentioned: it is quite sensible to not always translate a buffer object to an SSBO, because they require a very high GLSL version and more advanced GPU features.

4.2.9 Structures

Data structures are an essential component in almost every programming language. And in shading languages, too. There is even a programming paradigm named *structured programming* in which data structures, as the name implies, play a key role. The syntax of data structures is the same among all shading languages that are derived from the C programming language. Though, in addition to the syntax, HLSL provides a few more features for structures: *empty-* and *embedded* structures. These features are not supported in GLSL so we need to transform these structures appropriately. Advanced features like inheritance and member functions are examined in section 4.2.10.

Empty Structures

Structures can be empty in HLSL, i.e. they can have zero fields like `struct S {};` where no declarations are inside the braces of `S`. A variable with a type specifier of such a structure can even be initialized like any other structure, although there are no fields that could be initialized. This is not allowed in GLSL, neither by grammar nor by context. Although empty structures are probably rarely used in practical shaders, as long as the language supports them we need to consider their translation in our compiler. Fortunately, this can easily be solved by inserting a 'dummy' field into the structure. Since this field will never be used in the shader a sophisticated GLSL compiler and/or graphics driver will most likely remove it during the optimization passes. The example of the structure `S` from above would become something like `struct S { int dummy; };`.

Embedded Structures

Embedded structures (sometimes referred to as *nested structures*) are declared inside other structures and are only supported in HLSL up to SM 5. They require a little more effort because we need to pull declarations out of other declarations by operating on the AST. Furthermore, they can only be declared anonymously and must therefore be used in conjunction with a field declaration. Those anonymous structures are actually supported in GLSL, too, but since we need to move these structures out of another structure they must be uniquely identifiable. Those transformations on the AST can be done quite well within a separated pass *after* the semantic analysis but *before* the code generation. An illustration of such a translation is shown in Listing 4.26 and Listing 4.27.

Listing 4.26: Nested Structure in HLSL

```
1 struct S {
2     struct {
3         int InnerMember;
4     } OuterMember;
5 };
```

Listing 4.27: Nested Structure in GLSL

```
1 struct Anonymous1 {
2     int InnerMember;
3 };
4 struct S {
5     Anonymous1 OuterMember;
6 };
```

In this example, `S` specifies the "outer" structure and `OuterMember` specifies its field of a type with an anonymous structure (i.e. the "inner" or rather embedded structure). This anonymous structure is then renamed to `Anonymous1` in the resulting GLSL code, to be used as type specifier for the field declaration. The numbering is meant to point out that the cross-compiler needs to be aware of all auto.-generated names in which indexing is the most effective solution. Although the modification may look non-trivial when we only compare it textually, working on the DAST simplifies it considerably. This transformation is

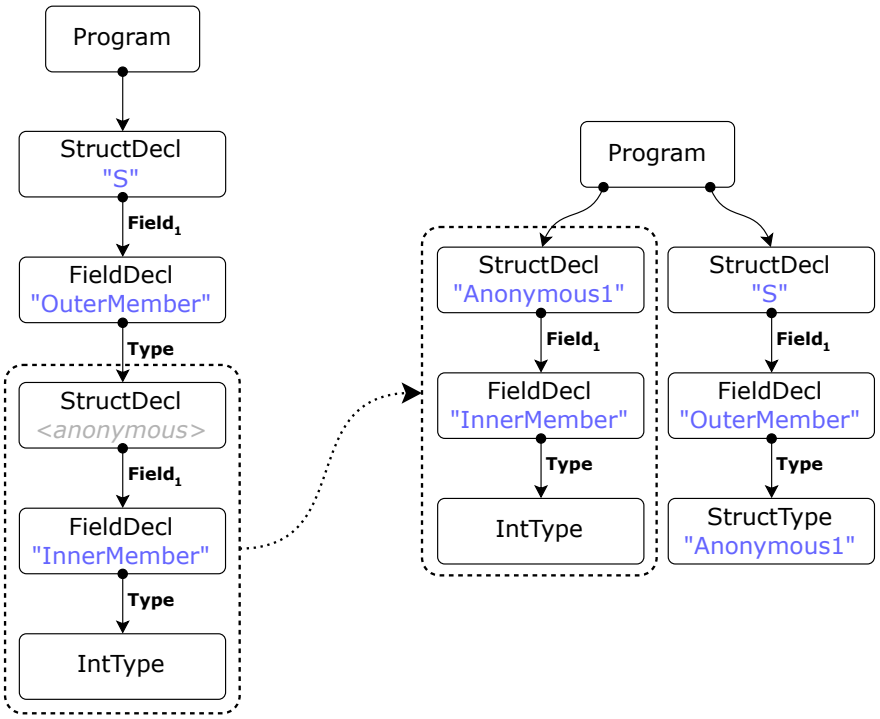


Figure 4.1: Simplified DAST of an embedded structure and its transformation.

illustrated in Figure 4.1 with a simplified DAST. Note that the example shows a DAST rather than an AST. This is because the information of the structure type "`Anonymous1`" is decorated to its declaration after

the transformation. Hence, after moving AST nodes around the referencing of types to their declarations must be updated correctly.

4.2.10 Member Functions

The last advanced feature of HLSL we examine here are member functions. From now on, we will use the term *class* when referring equally to classes and structures. In HLSL, structures and classes are equivalent anyway. These member functions are part of a class declaration and belong to the O-O paradigm. This feature also involves class inheritance[1] but not multiple-inheritance. Unlike other O-O programming languages such as C++ and Java, HLSL does not support *private*, *public*, or even *protected* access specifiers[2]. These words are reserved in the language for future use but using them in a shader results in a compile-time error. Since GLSL does not support member functions we need to transform all of these functions as well as their function calls into global functions. The member functions can of course access the data fields of its class. To determine whether an expression refers to a member field or a local variable, the analysis of member functions must be integrated into the type system. Just like in C++ and Java, there is the keyword `this` in HLSL which refers to the instance of the class on which the function was called. This keyword can only be used inside a member function to explicitly access a class member (either a field or function). Also members from a base class within the inheritance hierarchy can be accessed where the syntax is derived from C++.

Before we look at an example shader with member functions we first need to understand how the inheritance can be translated to GLSL. The naive approach is to insert all data fields of the base classes into the derived class. This is illustrated with a structure `B` which inherits from structure `A` in Listing 4.28 and 4.29.

Listing 4.28: Inheritance in HLSL

```
1 struct A     { float a; };
2 struct B : A { float b; };
```

Listing 4.29: Inheritance in GLSL

```
1 struct A { float a;    };
2 struct B { float a, b; };
```

In the GLSL translation, the structure `B` has adopted the data field `a`. There are two issues with this approach: Firstly, derived classes can have members with the same name as their base classes, and second, it makes the translation of member functions more difficult. The former means that `B` could have a field which is also named `a`, which would result in an invalid GLSL code when both fields have the same name. The latter means that all member functions of base classes must be translated multiple times or rather once for each derived type. A more advantageous approach is to insert the base class as first data field into the derived class. This data field needs a unique name which does not overlap with any other data fields. The previous GLSL example of structure `B` could then look like this: `struct B { A base; float b; };`. This approach naturally causes that each expression referring to such a base member needs to be transformed, too. Depending on the hierarchy depth, the expressions can become quite long. Nevertheless, this approach seems to be an elegant solution as we will see later in this section.

Let us consider the member functions again. Similar to embedded structures we also need to move all member functions out of a class declaration. To be able to access any member fields we also need to insert a special parameter into the parameter list of all member functions. Since the name `this` is reserved for future use, even in GLSL, we need to find another reasonable name. We can either use a cryptic name that is most likely never used, or we adopt the name `self` from other programming languages such as Python to make the output code as readable as possible. Just like with the `base` data field, we also need to insert this `self` parameter into all expressions referring to any member field. This can even create expressions in which both names appear. The difference by moving member functions out of the class declaration compared to embedded classes is that the functions must appear *after* the class declaration.

[1] *Inheritance* allows a class to be derived from another class.
[2] *Access specifiers* are used to limit the scope in which fields and functions can be accessed.

Otherwise the `self` parameter could not have the correct type, which must be the class for which it is declared. Next we take a look at a shader example that makes use of a member function, inheritance, and different kinds of member access. We will then review each part of the example from Listing 4.30 and Listing 4.31.

Listing 4.30: Member Functions in HLSL

```
1  struct Light {
2    float3 color;
3  };
4
5  struct PointLight : Light {
6    float3 pos;
7
8    float3 Shade(float3 P, float3 V) {
9      float3 L = normalize(P - this.pos);
10     float NdotL = max(0.0, dot(L, V));
11     return color * NdotL;
12   }
13 };
14
15 float4 ShadeScene() {
16   PointLight L;
17   L.color = (float3)1;
18   L.pos = (float3)0;
19   return L.Shade(WorldPos, ViewRay);
20 }
```

Listing 4.31: Member Functions in GLSL

```
1  struct Light {
2    vec3 color;
3  };
4
5  struct PointLight {
6    Light base;
7    vec3 pos;
8  };
9
10 vec3 Shade(
11   PointLight self, vec3 P, vec3 V)
12 {
13   vec3 L = normalize(P - self.pos);
14   float NdotL = max(0.0, dot(L, V));
15   return self.base.color * NdotL;
16 }
17
18 vec4 ShadeScene() {
19   PointLight L;
20   L.base.color = vec3(1);
21   L.pos = vec3(0);
22   return Shade(L, WorldPos, ViewRay);
23 }
```

In Listing 4.31 we can see the insertion of the `base` data field in the structure `PointLight` as well as the parameter `self` in the function `Shade`. The `self` parameter is used twice: First, as replacement of the keyword `this` in line 13, and second, to access the `color` data field in line 15. The latter one also needs the insertion of the `base` data field since it belongs to the base structure `Light`. The object expression `L` in line 22 can simply be moved as first argument into the function call of the member function. When the member function belongs to a base class, we need to insert the respective `base`-field expressions again. This is the case in line 20 for instance, where the data field `color` is initialized. The content of the functions in this example only perform very basic lighting effects. This approach works quite well even when the same function name is used in the global scope, due to function overloading[1]. Nevertheless it is useful to attach the class name to their member functions in the translation, just to make sure a conflict in function overloading is impossible. The member function in this example could then be named `PointLight_Shade` for instance. Otherwise the user could get an interface to either enable or disable this kind of renaming. A similar approach was implemented in the ancestor of the C++ programming language back in 1979 when its predecessor was named “C with Classes” [52]. The source code was compiled into C instead of assembler directly and member functions were translated into global functions which are compatible with the C language.

4.2.11 Boolean Vector Expressions

There are a few special intrinsics for Boolean vectors in GLSL that we have not covered so far. The functionality of these intrinsics is also supported in HLSL but they are part of the expression syntax instead. They are used to compute Boolean expressions for vectors such as $a < b$ where $a, b \in \mathbb{R}^4$. In

[1] *Function overloading* refers to a feature that a function name is used multiple times, but the compiler can still determine which function is used as long as the functions can be distinguished by their parameter lists.

HLSL, this vector comparison results in a 4D Boolean vector where each component has either the value *true* or *false* and the *less-than*-comparison is performed per-component. The same effect is achieved with a `lessThan(a, b)` intrinsic function call in GLSL, and there is an intrinsic for each Boolean operator. The mapping between the operators in HLSL and the intrinsics in GLSL is shown in the following table:

HLSL	==	!=	<	<=	>	>=
GLSL	`equal`	`notEqual`	`lessThan`	`lessThanEqual`	`greaterThan`	`greaterThanEqual`

Note that these operators/intrinsics cannot be used as conditional expressions, i.e. they cannot appear inside the condition of an `if`-statement for instance. Instead, they can be used to initialize a Boolean vector (e.g. `bool4` in HLSL or `bvec4` in GLSL). However, there is one exception for conditional expressions in HLSL, which is the so-called *ternary operator*[1]. The ternary operator allows selecting its result between the two branches per-component. This can be achieved with the `mix` intrinsic in GLSL and is illustrated in Listing 4.32 and Listing 4.33.

Listing 4.32: Boolean Vector in HLSL

```
1 float4 a = float4(1, 2, 3, 4);
2 float4 b = float4(4, 3, 2, 1);
3 bool4 condition = a < b;
4 float4 x = condition ? a : b;
```

Listing 4.33: Boolean Vector in GLSL

```
1 vec4 a = vec4(1, 2, 3, 4);
2 vec4 b = vec4(4, 3, 2, 1);
3 bvec4 condition = lessThan(a, b);
4 vec4 x = mix(a, b, condition);
```

From the example above, the resulting vectors are $condition = (\text{true}, \text{true}, \text{false}, \text{false})^T$, $x = (1, 2, 2, 1)^T$ for both GLSL and HLSL. In this case we don't even have to handle duplications of any expressions which makes the transformation on AST level quite easy to implement. However, it is important that the tree structure of binary expressions is correct in the AST, taking into account the operator precedence. Otherwise, an expression like `a < b + c` might be erroneously translated to `lessThan(a, b) + c` instead of `lessThan(a, b + c)`. If GLSL supported the same syntax as HLSL for Boolean vectors, the correct operator precedence would be less important, since the cross-compiler would produce the same output independently of the AST hierarchy.

Apart from Boolean vectors, it should be mentioned that only GLSL has a *Logical-Exclusive-OR* operator (also *Logical-XOR*, but not to be confused with *Bitwise-XOR*) to compare two Boolean values. If we have those declared as `bool P,Q;` the comparison `P ^^ Q` (Exclusive-OR) in GLSL is equivalent to `P != Q` (Inequality) in HLSL. That this is indeed the case, shows the following truth table:

P	**Q**	**P != Q**	**P ^^ Q**
F	F	F	F
F	T	T	T
T	F	T	T
T	T	F	F

4.2.12 Name Mangling

We already mentioned briefly the process of name mangling in a previous section. Now we are going to discuss at which point name mangling is required. The general idea is simply to append a prefix (or suffix) to the name that is written to the output code to avoid duplicate identifiers or reserved words. Various programming languages use the prefix of two consecutive underscores for their internal name mangling. GLSL for instance is one of those languages where identifiers beginning with two underscores are reserved, i.e. declaring a variable like `__example` will result in a compile error. The easiest way to find

[1] *Ternary operator* consists of three expressions: a condition, a *then*-branch, and an *else*-branch; It is usually written with the `?:` characters.

a valid prefix is to let the user of the cross-compiler choose. This is a contract between the user and the compiler that no identifier in the code will begin with that prefix. Otherwise the shader cross-compiler can report an error.

Name mangling is at least required for reserved words such as intrinsics, built-in variables, and keywords. For auto.-generated/temporary variables, it is more of a recommendation rather than a requirement. But anonymous structures for instance that have been renamed should also be taken into account for name mangling. If our name mangling prefix is defined to be "`NM_`" for instance, a function named `main` in HLSL could be renamed to `NM_main` in GLSL. The process of name mangling must be performed at the right place, otherwise a translated intrinsic might be erroneously renamed. For example, there is an intrinsic to compute the fractional part of an argument which is named `fract` in GLSL and `frac` in HLSL. So declaring a function named `fract` is valid in HLSL but must be renamed in GLSL, while a call to `frac` in HLSL is valid, too, it must be renamed to `fract` in GLSL without name mangling. As we have seen earlier, the renaming of member functions can be part of the name mangling process, too.

4.3 Restrictions

After examining many different language features, we will now summarize the restrictions we have gathered. We have seen various language features that can be translated straightforward such as constant buffers and matrix ordering. We have also seen that an elaborated type system is absolutely necessary to provide a solid basis for a source-to-source compiler. Compromises are sometimes unavoidable as the tessellation-shader-related attributes have shown. In those cases an appropriate interface for the user has to be provided. This in our case could be a command line tool with corresponding commands and/or a software library with the respective input parameters. Especially the translation of the IO semantics is challenging. In combination with embedded structures and multiple uses of IO structures the HLSL→GLSL translation needs an extensive code analysis to work properly. Moreover, some restrictions related to sampler states must be accepted, at least when the GLSL shaders are targeted to OpenGL rather than Vulkan.

There are a few language components we have not covered: *techniques* from older HLSL versions, and *submodules* from GLSL. Both components are obsolete. Therefore it is justifiable to ignore them. We will briefly mention them here anyway. *Techniques* were used in so-called *effect files* to specify how a collection of shaders is meant to be compiled and executed. This is primarily done on CPU-side but it was a convenient way to specify it within the shader source file. It would be possible to develop a software interface which allows the user of our cross-compiler to query all techniques of a shader so the user can transfer them to the respective OpenGL commands. But since techniques are outdated nowadays, we can safely ignore them. Hence, our compile will parse them just like commentaries whenever they appear in a shader. We can do the same for *submodules* in GLSL. They even have been removed from the GLSL specification for Vulkan. Furthermore, they were widely rejected by shader programmers, among other things because of their disadvantageous overlap with uniforms. They have a similar purpose as the *dynamic linkage* in HLSL with classes and interfaces. It is meant to provide a mechanism to dynamically change the behavior of the shader to eliminate the need for so-called *uber shaders*[1]. Since the translation of submodules into interfaces/classes and vice-versa requires a high effort, and the fact that this language component is obsolete, we can ignore it with good reason.

[1] *Uber shaders* combine multiple purposes with static compile-time conditions that are commonly implemented with macro expansion, which makes those shaders hard to read and maintain.

5 Implementation

In this chapter we will analyze in more depth how a practical source-to-source compiler for shading languages can look like. We will focus on various implementation details and see how difficulties can be circumvented. A prototype for this thesis, as a proof-of-concept, is available at `github.com` in form of the XShaderCompiler project [14]. A basic overview of the compiler infrastructure is illustrated in Figure 5.1. In the following sections, we will analyze these compiler stages in detail. Since the XShaderCompiler

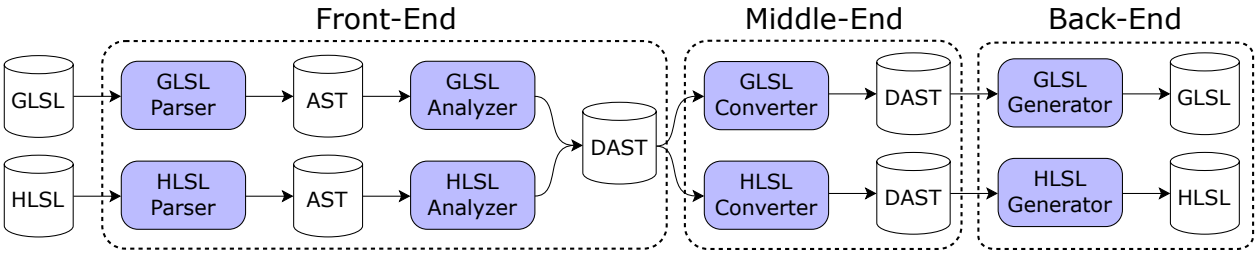


Figure 5.1: Basic graph of the XShaderCompiler; White ≙ data models, Blue ≙ compiler stages.

is only a prototype, some compiler stages might be incomplete. However, the compiler is already able to perform HLSL→GLSL translations properly. In contrast to the usual compiler pipeline, the *middle-end* does not contain optimization passes. Instead, it includes the part of converters to transform the AST into the target language. This is where new AST nodes and temporary variables are generated, others are removed and reconnected. We will now go through all stages of this pipeline step-by-step.

5.1 Front-End

5.1.1 Uniform AST Design

Since the AST is a fundamental structure wherewith the compiler needs to work throughout the entire infrastructure, its design is crucial. Since we do not use an IR but still want to support multiple source and target languages, it is important to have a unified AST design. As a result, we can neither use a parser generator nor derive the AST classes directly from the grammar of any shading language. We need to reduce our hierarchy to a common denominator which is compatible with all provided shading languages, i.e. GLSL and HLSL in our case. Fortunately, GLSL and HLSL have a very similar syntax and are both derived from the C programming language. Among a few other things, the major differences are their different sets of keywords. There will be some AST classes which need to be defined exclusively for either one of both languages. But most of the AST classes can be shared between them. If we were also supporting MSL, then we would have to rethink this approach. This is because MSL is derived from C++14 which is a very extensive programming language. Among other things, the type system would need to be far more complex, due to the C++ templates. A compiler for such a complex language is often based on the Clang/LLVM framework. But since we are limited to GLSL and HLSL, we can certainly develop our own one.

First of all, we need to classify our AST classes. This will specify the class inheritance as well as the base classes. `AST` is the name of the root, i.e. each AST class is a sub class (or sub-sub class etc.) of `AST`. The major base classes are for *statements* and *expressions*. An expression is usually a combination of several sub-expressions. For example, `1+2` is a *binary expression* with operator `+` and the two *literal expressions* `1` and `2` as sub-expressions. A statement usually stands for its own and may contain multiple

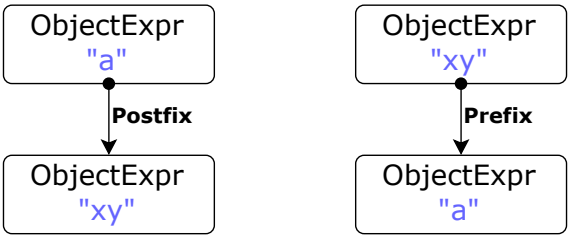
sub-expressions, while an expression naturally cannot contain something like a 'sub-statement'. In many programming languages derived from C, a statement is most of the time terminated by a semicolon, which is called the *statement terminator*. Other programming languages use the *new-line* character as statement terminator. Let us now divide our AST classes into these four categories:

- **Statement** (abbr. Stmnt): Common control flow and collection of object declarations.
- **Expression** (abbr. Expr): All kinds of expressions, e.g. arithmetic, function calls, assignments etc.
- **Declaration** (abbr. Decl): Object declaration such as variables, buffers, textures etc.
- **Miscellaneous**: All remaining AST classes such as array dimension, code blocks, program root etc.

The reason why we gave declarations an own category is that almost every object declaration can contain multiple named objects within a single statement. For example, `float a,b;` declares two objects, namely "a" and "b", which are contained within a single variable declaration of type `float`. The single declarations (i.e. Decl) will later be used for *decorating* the AST during the context analysis. In this way, the DAST can have references (or rather pointers) to the `Decl` base class. For example, in an assignment expression like `a+=42` the respective expression class must have a reference to a `Decl` object for "a", otherwise the compiler cannot determine, if "a" was declared as a variable on which such an operation is allowed. There is another base class (or rather an interface) we have not mentioned yet, because it is more a helper class for simplification rather than a category. We will call this class `TypedAST` which holds a *type denoter* and is used for each AST class which can return a type. Its base class is `AST` and its sub-classes are `Expr` and `Decl`. In this way, we can query the type of each expression and declared object. For example, if "a" was declared as a floating-point variable, its `Decl` object will return the respective type denoter for floating-points.

Expressions

Let us now focus on the design of our expression classes. Expressions can be quite complex in many programming languages, as well as in shading languages. There are even programming languages in which (almost) everything is considered being an expression and thus yields a value, like in Scala or Haskell. Statements like the loop-statements, in contrast, are easy to parse and also easy to analyze. This is because they have a distinct meaning and they also have their own keywords which makes it easy to identify them during the parsing process. The meaning of expressions, however, highly depends on the context and the grammar is generally overloaded. A sophisticated design for expression classes is also desired for performance reasons. If we need to traverse a sub-tree many times only to access a simple variable within an expression, we would waste a lot of computation time. The tree hierarchy for a binary expression is self-explanatory but for object- and sub-object-expressions there are two basic approaches: we can either define a *postfix* or a *prefix* sub-expression. Consider a vector $a \in \mathbb{R}^4$ whose first two components are accessed in an expression like this: `a.xy`. This expression consists of two sub-expressions where both refer to an object: 'a' and 'xy'. If we were to use *postfix* expressions the tree hierarchy could look like shown in Figure 5.2a. If we were to use *prefix* expressions instead, the tree hierarchy might look as shown in Figure 5.2b. In both cases the expression class `ObjectExpr` will have the same fields (only different names were used). The difference is in the construction of the AST and thus in the way of traversing it, too. Choosing between the two approaches affects further classes. We can apply the same hierarchy for an array index expression, such as `a[0]`. In this case we would end up with two classes like `ObjectExpr` and `ArrayExpr`. Both classes need either a postfix or prefix field of type `Expr`, i.e. the base class of all expressions. The question now is which approach is the better one. Recall that traversing the AST is a frequently used task during the entire compilation process. Although the type information depends on the entire expression tree, only the rightmost expression holds the type we are interested in. Therefore, the **prefix** approach is an elegant and efficient solution, although it may look extraordinary, due to the reverse order of identifiers in the hierarchy. For the sake of clarity, consider the example from Figure 5.2b again. Here our topmost AST node is the object we actually want to access in the



(a) Postfix Traversal: `a.` → `xy` **(b)** Prefix Traversal: `a` ← `.xy`

Figure 5.2: AST object expression examples with postfix and prefix construction/traversal.

expression `a.xy`. And this node already holds the type information we need, assumed that the type has already been deduced during the analysis. In the other case, shown in Figure 5.2a, we would first need to traverse to the next sub-expression to access the object we are looking for. This obviously exacerbates with each new hierarchy level. It is also advisable to follow the Divide-and-Conquer (D&C) principle. That is, we will divide all expression classes into small pieces, each with a single functionality. For example, the `ArrayExpr` class will only hold a list of array indices and a prefix expression, nothing else. For a better understanding of the AST design for expressions consider another example. This time, we will use several types of sub-expressions to be sure that we cover all sorts of expressions that are possible in our shading languages. Especially the concatenation of sub-expressions is quite interesting. Since only HLSL supports member functions and has thus a little more extensive expressions, we will take an example from that language. An extensive expression example is shown in Listing 5.1.

Listing 5.1: Extensive expression in HLSL

```
1 //struct    func              field      func          comp
2 ( Scene::getLights() )[7].material.fadeColor(0.3).rgb
```

All identifiers are marked with their meaning: Structure (*struct*), function (*func*), structure field (*field*), and components (*comp*). All these identifiers *can* be covered by the `ObjectExpr` AST class. However, the function calls, namely `CallExpr`, will have their own identifier. Although this is against our D&C principle, it will make the traversal easier and since dynamic function objects do not exist in our shading languages, we don't need to handle them as object expressions. If we were supporting function pointers, as the C programming language does, it would be sensible to cover all function names with the `ObjectExpr`. Hence, the call expressions hold a list of arguments, an identifier for the function name, and again a prefix expression. Static functions must be concatenated with `::` instead of `.`, which can be stored in a Boolean field of the `CallExpr` class, like $isStatic \in \mathbb{B}$. The resulting AST hierarchy is illustrated in Figure 5.3, which shows pretty clearly the advantage of the prefix approach: The entire expression ends in the access of the `rgb` components which results in a 3D-vector type that can be buffered in the AST node. If this expression were assigned to another 3D-vector, our compiler can check very quickly if the assignment is valid. All the types in between this expression are irrelevant for this type check. They are only necessary during the first time the type of each sub-expression is deduced. In addition, the traversal is always from right-to-left. Mixing the two approaches should be avoided in principle. For other expres-

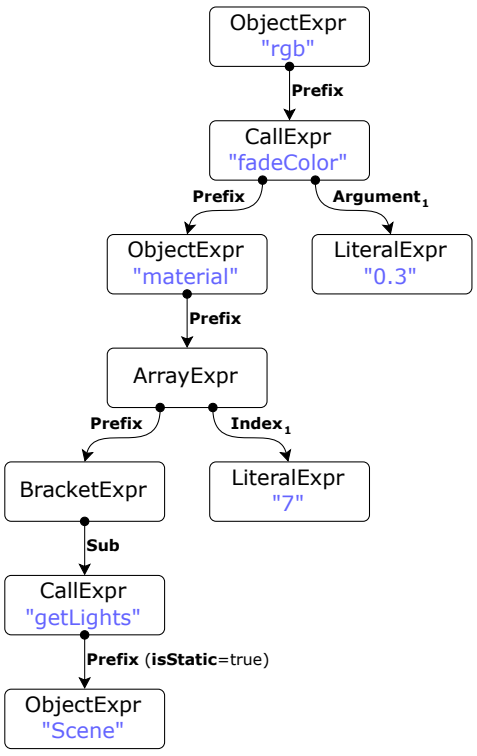


Figure 5.3: AST of the extensive expression example in HLSL.

sions and especially the binary expression, namely `BinaryExpr`, the order of traversal is determined by the language specification. An arithmetic expression like `1-2*3+4` can of course not be traversed simply from right-to-left. It depends on the *operator precedence* as well as on the *associativity*.

Class Hierarchy

We will now summarize the entire AST class hierarchy and discuss certain classes specifically. For an outline of all AST classes, take a look at Figure 5.4. All classes in blue are only provided for GLSL, and all

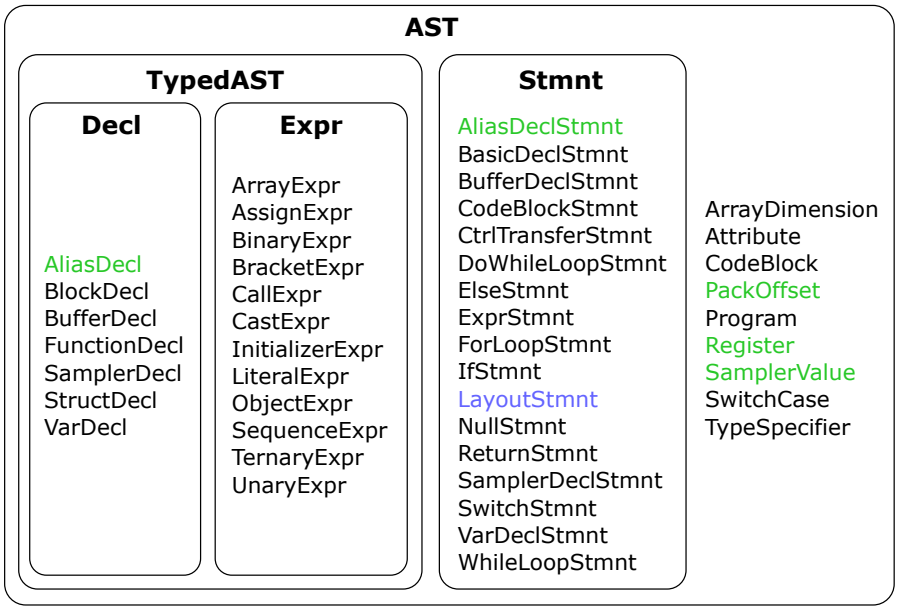


Figure 5.4: Outline of the AST class hierarchy; Blue ≙ GLSL only, Green ≙ HLSL only.

classes in green are only provided for HLSL. Any other classes are shared between both languages. Some of them may have various data fields that are only used in either one of those languages, though. This hierarchy is just a proposal but almost identical to the implementation in the XShaderCompiler project. There are a few additional AST classes that are only used for simplification during the parsing process. It is also legitimate to combine more classes like the loop statements for instance. A too small set of classes, however, could make the handling more difficult.

Let us briefly discuss a couple of specific classes. `AliasDecl` and `AliasDeclStmnt` are for type aliasing via the `typedef` keyword in HLSL. Just like variable declarations, an alias declaration can have several members within a single statement, too. For example, `typedef int Int1,Int2[2];` results in two instances of `AliasDecl`, one for `Int1` of type *integer* and another one for `Int2` of type *array-of-integer*. The classes `PackOffset` and `Register` are for the HLSL keywords of the same name. For GLSL, this offset and register information is stored inside a layout attribute, which is covered by the `Attribute` class. In addition, the `SamplerValue` class is only for the entries of a *static* sampler state declaration, i.e. when the attributes of a sampler state are specified within the shader program. This is only supported in HLSL, while in GLSL the sampler states can only be configured by the runtime system on the CPU. The `LayoutStmnt` class is for global layout qualifiers like we have seen in Listing 4.15 for instance. They do not belong to a variable declaration but for the global shader itself. The interested reader may have noticed the class `ExprStmnt`, which may seem strange at first sight. It is used for all expressions that occur as standalone statements such as function calls or variable assignments. Pointless statements like binary operations that are not assigned to anything (e.g. `;1+2;`) are allowed as well which are also covered by this class. The class `BasicDeclStmnt` is the default class for all single declaration objects: functions,

structures, and block declarations, where the latter one covers constant buffers/UBOs and SSBOs. All remaining AST classes are more or less self-explanatory. The `UnaryExpr` class could also be split up into a *pre-* and *post-* unary operator. For example, one for `++i` and one for `i++`. But keep in mind that an increasing amount of AST classes will increase the effort of maintainability throughout the entire compiler infrastructure.

Now recall that we mentioned earlier that it might be quite reasonable to use a CST instead of an AST for a cross-compiler. This is because we want the code structure as well as commentaries to be preserved in the output code. However, we still use an AST since it is much easier to handle than a CST. Especially *white spaces*, i.e. blanks, tabulators, and new-line characters, make the parsing process even more complicated. Nevertheless, we want to preserve the commentaries at least. We can do this by adding a list of strings as data field into the `Stmnt` base class. This list will store all commentary lines that appear in front of a statement. All other commentaries like the one inside an expression will be ignored. This is a reasonable compromise since most commentaries that are used for documentation are written in front of a statement like a function declaration.

5.1.2 Generic Parsing

To follow the Don't Repeat Yourself (DRY) principle and avoid redundant code, we have to split the parser into several classes, each of which performs a certain task of the parsing process. Even the preprocessor is part of this class hierarchy, since it requires to parse and evaluate constant expressions like the `#if`-directive. Several language constructs such as the loop statements and almost every expression can be moved into a base class, too. A proposal for a class hierarchy of parsers is shown in Figure 5.5. For our

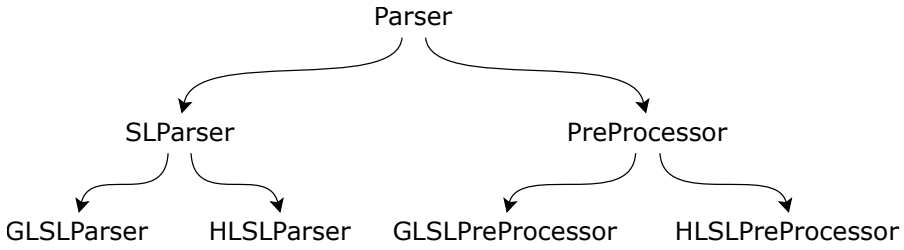


Figure 5.5: Outline of the Parser class hierarchy.

cross-compiler, an LL(1) parser is sufficient to process both GLSL and HLSL source code. That means our parser can only look ahead one token to derive the AST. The root class `Parser` contains the basic functionality like error handling, accepting tokens from the token scanner, and also expression parsing which can be treated equally in all sub classes. The `SLParser` (for "Shading Language Parser") is the base class for the GLSL and HLSL parser and covers the parsing of common statements like code blocks, loops, return statement etc. The same abstraction is applied for the preprocessors. However, the base class `PreProcessor` can handle almost every preprocessor directive, since its sub-language is almost identical between GLSL and HLSL. Hence, `GLSLPreProcessor` for instance has only a few additional directives to process such as the `#version`-directive. A similar class hierarchy for the token scanners is recommended.

Non-Context-Free Parser

The grammar as well as the parser for our shading languages are usually context free. However, there is one syntactic ambiguity within HLSL, which is the *cast expression*. We have already seen various examples of cast expressions but never mentioned what other expressions can look equally. An example of this situation is shown in Listing 5.2 and Listing 5.3.

Listing 5.2: Cast Expression in HLSL

```
1 int MinusOne() {
2     // Equivalent to (int)-1
3     typedef int X;
4     return (X) - (1);
5 }
```

Listing 5.3: Non-Cast Expression in HLSL

```
1 int MinusOne() {
2     // Equivalent to 0-1
3     int X = 0;
4     return (X) - (1);
5 }
```

We have two variants of the function `MinusOne`, which returns the value −1 in two different and cumbersome ways. In line 4, the expression of the return-statement of both examples looks the same. Although they return the same value, it does not mean the same. The first variant from Listing 5.2 is a *type cast* from the *unary-expression* `-(1)` to the type named `X` (via type aliasing). The second variant from Listing 5.3 is a *subtraction binary operation* between the value of the local variable `X` and the constant value `1`. Both functions will return the value −1, but the parser must create two different ASTs. However, without the context information what `(X)` denotes, the parser is not able to generate the correct AST. Alternatively, the parser could always generate one of those variants and the compiler must then transform the AST during the context analysis. But a simple and effective solution is to integrate a symbol table into the parser. This symbol table keeps track of all type names within the current scope. The parser can then determine whether the name inside parentheses is a type name or not. As a result, we no longer have a context-free parser. But it is a good trade-off between specification and simplicity. In a programming language more complex than HLSL this simple solution would probably fail. This is because in programming languages that don't require forward declarations, types can be used before they were defined.

Bidirectional Keyword Mapping

In a very basic cross-compiler that does not perform any contextual analysis we could simply store the input keywords and map them directly to the respective output keywords. For example, for a call expression of the intrinsic `fract` in GLSL, we could translate this name directly to `frac` in HLSL using a hash-map or any other kind of lookup table. But since we are developing a more sophisticated cross-compiler that performs various transformations on the AST, we need to work with these keywords more than once. If we were using the keywords in our AST directly, we would have to make a lot of string comparisons. Moreover, dealing with strings all the time is very unhandy and generally bad practice in computer programming, due to potentially unrecognized typing errors that even the compiler cannot detect. Therefore, we translate the keywords into constant integrals from a fixed set of enumerations. We can then operate on these enumerations and translate them back into the target keyword during the code generation. These enumerations could be described as a kind of IR, although this term is actually reserved for another scope. Furthermore, there are various keywords which have the same meaning, especially in HLSL. For example, `bool`, `bool1`, and `bool1x1` are keywords for the same Boolean type in HLSL, just in scalar, vector, and matrix notation. To avoid unnecessary enumeration entries, we have to map those keywords to the same value. As a result, our *string-to-enumeration* mapping function $f_{S\to E}: \Sigma \to \mathbb{N}$ is **surjective**, and our *enumeration-to-string* mapping function $f_{E\to S}: \mathbb{N} \to \Sigma$ is **injective**. To recall the definition of surjectivity and injectivity, see Equation 5.1.

$$\begin{aligned} &\text{If } f: X \to Y, \text{ then } f \text{ is said to be } \textbf{surjective} \text{ if } \forall y \in Y, \exists x \in X, f(x) = y. \\ &\text{If } f: X \to Y, \text{ then } f \text{ is said to be } \textbf{injective} \text{ if } \forall a, b \in X, f(a) = f(b) \Rightarrow a = b. \end{aligned} \tag{5.1}$$

The mapping from enumeration to string is required for the code generation where we only need one keyword. This is why the function $f_{E\to S}$ is injective or rather we have more keywords than enumeration entries. Both functions can be defined inductively, as shown in Equation 5.2. With such a bidirectional mapping functionality, we can easily define our sets of keywords and map them efficiently between our source and target languages. In order to better understand the performance benefit, consider the

following example: when we have a keyword that we need to compare against a set of N strings, each of which has an average length of M characters, we would need to make roughly $N \times M$ character comparisons in a worst case scenario.

$$\begin{aligned} f_{S\to E}(\text{“}\texttt{bool}\text{”}) &= 0 & f_{E\to S}(0) &= \text{“}\texttt{bool}\text{”} \\ f_{S\to E}(\text{“}\texttt{bool1}\text{”}) &= 0 \\ f_{S\to E}(\text{“}\texttt{bool1x1}\text{”}) &= 0 \\ f_{S\to E}(\text{“}\texttt{int}\text{”}) &= 1 & f_{E\to S}(1) &= \text{“}\texttt{int}\text{”} \\ \cdots & & \cdots \end{aligned} \tag{5.2}$$

In contrast, the enumeration can be associated in constant time. This is a complexity improvement from $O(N \times M)$ to $O(1)$, in the *Landau Notation* which is also referred to as *Big O Notation* [25].

5.1.3 Context Analysis

The context analysis is the phase in which all associations between AST nodes are established. This phase must also detect all typing errors by checking if every used identifier has been declared according to the language specification. The core of this analysis is the symbol table which must also support function overloading, i.e. the association of an identifier to multiple instances. The symbol table that we have used for the parser may be less extensive, since it is only used to check type names, which cannot be overloaded.

Overloadable Symbol Table

We will now describe the mode of operation of our symbol table in detail. The symbol table has to provide four basic functions which we will denote as the following:

- Open new scope for subsequent symbols.
- Close current scope and release all symbols that are associated with that scope.
- Register new symbol and associate it with a specified identifier, or emit error on failure.
- Fetch the symbol that was associated with a specified identifier, or emit error on failure.

The functionality of overloading is not handled by the symbol table itself, or at least not directly. Instead, the symbols that are stored inside the symbol table can have multiple references. The Fetch function will only return such a symbol, and if it's overloaded, a further analysis must be done. Otherwise, the error of an ambiguity must be reported. The basic idea of these four functions is illustrated in Algorithm 2. For this algorithm, let $T_{\Sigma\to\mathbb{S}}$ be a lookup table that maps identifiers to stacks of symbols, let $T_{\{\Sigma\}}$ be a stack of lists of identifiers, and let Ω be the set of symbols that can be overloaded. For an identifier i, our lookup table $T_{\Sigma\to\mathbb{S}}$ provides the symbols for all identifiers in a bottom-up fashion. That is if the identifier i is used multiple times but in different nested scopes, the lookup table $T_{\Sigma\to\mathbb{S}}$ provides the symbol from the deepest scope. The stack $T_{\{\Sigma\}}$ is used to release all symbols from their respective scopes when they are closed. Moreover, the set of symbols that can be overloaded, namely Ω, generally only contains the symbols for function objects. As a result, the symbols in our symbol table are not the instances of AST objects, like `VarDecl` or `FunctionDecl` for instance. Instead, they are delegates for these AST nodes. A symbol for a function identifier such as “`VectorLength`” might have multiple references to the actual `FunctionDecl` AST nodes. For example, one for a 2D-vector and another one for a 3D-vector. Whether overloading is allowed or not for a given function object must be determined when the new identifier is about to be registered. The information for this is already stored inside the `FunctionDecl` class in form of the parameters and their type denoters. The final deduction rules to choose one of the references when a symbol is fetched depend on the source language. These deduction rules are very similar between GLSL

Algorithm 2 Basic Functions for Symbol Table

procedure OPEN($T_{\{\Sigma\}}$)
    $T_{\{\Sigma\}} \hookleftarrow \{\}$ ▷ Push empty list of identifiers onto stack
procedure CLOSE($T_{\{\Sigma\}}, T_{\Sigma\to\mathbb{S}}$)
    $\Sigma \leftarrow T_{\{\Sigma\}}(0)$ ▷ Get top most list of identifiers from stack
    for all $i \in \Sigma$ **do**
        $\mathbb{S} \leftarrow T_{\Sigma\to\mathbb{S}}(i)$ ▷ Find stack of symbols via identifier in lookup table
        $\mathbb{S} \hookrightarrow \epsilon$ ▷ Pop top most symbol from stack
        if $\mathbb{S} = \varnothing$ **then**
            $T_{\Sigma\to\mathbb{S}}(i) \hookrightarrow \epsilon$ ▷ Remove stack for identifier i from lookup table
    $T_{\{\Sigma\}}(0) \hookrightarrow \epsilon$ ▷ Pop top most list of identifiers from stack
procedure REGISTER($T_{\{\Sigma\}}, T_{\Sigma\to\mathbb{S}}, \Omega, i \in \Sigma, s \in \mathbb{S}$)
    if $i \in T_{\Sigma\to\mathbb{S}}$ **then** ▷ Check if identifier is already defined in current scope
        $s' \leftarrow T_{\Sigma\to\mathbb{S}}(i)$ ▷ Find previous symbol in lookup table
        if $s' \in \Omega$ **then**
            $s' \hookleftarrow s$ ▷ Overload identifier i by integrating s into s'
        else
            ERROR(i) ▷ Report error about duplicate use of identifier i
    else
        $T_{\Sigma\to\mathbb{S}}(i) \hookleftarrow s$ ▷ Append symbol for identifier i to lookup table
procedure FETCH($T_{\Sigma\to\mathbb{S}}, i \in \Sigma$)
    return $T_{\Sigma\to\mathbb{S}}(i)$ ▷ Find symbol in lookup table

and HLSL, but there are some additional rules for the latter one. In HLSL, a texture object can optionally be declared with a generic type for the texture elements, such as `Texture2D<float3> TexRGB;`. The default sub type is a 4D-vector, i.e. `float4`. GLSL has only very limited support for texture sub types and the dimension of the texture elements cannot be changed. Now consider an overloaded function whose instances take only one parameter each. For each instance, this parameter is a 2D-texture type and the functions can only be distinguished by the generic sub type of this texture parameter. Our cross-compiler must translate all of these parameter types to `sampler2D` (or `texture2D` for Vulkan). Hence, the information to distinguish the functions is lost. The only way to prevent this is that our compiler generates different names for these functions in the output code. This can also be integrated into the process of *name mangling*. An example of this translation is illustrated in Listing 5.4 and Listing 5.5.

Listing 5.4: Function Overloading in HLSL

```
1 float3 RGB(Texture2D<float3> tex) {
2     return tex.Sample(S, P);
3 }
4 float3 RGB(Texture2D<float4> tex) {
5     return tex.Sample(S, P).rgb;
6 }
```

Listing 5.5: Function Overloading in GLSL

```
1 vec3 RGB_1(sampler2D tex) {
2     return texture(tex, P).rgb;
3 }
4 vec3 RGB_2(sampler2D tex) {
5     return texture(tex, P).rgb;
6 }
```

In this example, `S` denotes a sampler state which is ignored in the GLSL output, `P` is a global position to sample the texture from, and `RGB` denotes the overloaded function.

Next, we will analyze how a function is deduced when its name is overloaded. A function in HLSL has a minimal and a maximal number of arguments. They only differ if the function has any parameters with *default argument* (sometimes misleadingly referred to as *default parameter*). That is, that some arguments in a function call are optional whenever their respective parameter has a default value. In the following, let Π_f be the set of all parameters of a function f, $\Delta_f \subseteq \Pi_f$ be the set of parameters with default argument, and A be the set of arguments that are given for a function call with the identifier of

f. Then $\omega_f = |\Pi_f|$ is the maximal number of arguments and $\alpha_f = |\Pi_f \setminus \Delta_f|$ is the minimal number of arguments. That is, $\alpha_f \leq |A| \leq \omega_f$ must hold true, otherwise the function call is invalid. Now let f_0, f_1 be two functions with the same identifier but with different parameter lists. There are basically two cases where an ambiguity in a function call can occur: Firstly, if $|A| = \alpha_{f_0} = \omega_{f_1}$ holds true, and secondly if all arguments can be implicitly assigned to the parameter lists of all function candidates equally. In these cases, the compiler must report an error since the argument list does not fit to any given function candidate. Both cases of ambiguity are illustrated in Listing 5.6.

Listing 5.6: Ambiguous Function Calls in HLSL

```
1 void First(int x);
2 void First(int x, int y = 3);
3 void Second(float x);
4 void Second(double x);
5 /* ... */
6 First(1);  // Ambiguous function call, due to default argument of parameter 'y'
7 Second(2); // Ambiguous function call, due to type mismatch (int vs float/double)
```

Now that we know how those ambiguous function calls can occur, we can focus on the part that deduces the functions where no ambiguity exists. This algorithm can be summarized into the following steps, assumed that we already have the list of function declarations with the same identifier from our symbol delegate:

1. Match number of arguments to function declarations.
2. Find function candidates:
   a) Find candidates with explicit type matching, or ...
   b) ... find candidates with implicit type matching otherwise.
3. Return deduced function or emit error if number of deduced candidates is $\neq 1$.

In the first step, we check if there is any function whose number of parameters matches to the number of arguments. This can be done by applying our formula from above, namely $\alpha_f \leq |A| \leq \omega_f$, until we find a function for which this formula holds. The function candidates are determined by matching the argument types to their parameter types. This functionality can be implemented within the type denoter classes and should provide an explicit and implicit type matching. The explicit type matching must compare the types for equality, and the implicit type matching must compare the types for compatibility via type-cast. If only one candidate remains in the last step, the function has been deduced successfully. Otherwise, either no function could be deduced (like with `Second` from Listing 5.6), or more than one function has been deduced (like with `First` from Listing 5.6).

The last part for our symbol table is the structure members. This can be implemented directly inside the `StructDecl` AST class. The only functionality we need here is to fetch a member by its identifier from each structure. The registration of the identifiers can be processed during the parsing. As a result, we only need an equivalent of the FETCH function.

Type Deduction

In section 4.2.4 we have seen an outline of all classes for our type system. Now we will discuss how the type of an arbitrary expression can be deduced. This is done primarily by the interface of two functions: one for type denoters and the other one for the typed AST classes (i.e. all sub-classes of `TypedAST`). We will call the first function SUBSTITUTE with which we replace an input expression by the sub types of a type denoter. The second function is called DEDUCE which passes the references of the DAST to the SUBSTITUTE function. After DEDUCE has successfully deduced a type for an AST node, the function buffers the type so it does not have to deduce it again. DEDUCE is primarily used to combine multiple types with each other. This is why the AST classes with the most non-trivial implementations of this function are

`UnaryExpr`, `BinaryExpr`, `TernaryExpr`, and `CallExpr`. The latter one is discussed in the next section. For example, `BinaryExpr` needs to find a common type denoter from its two sub-expressions of the left-hand side and the right-hand side as well as its operator. All other typed AST classes will predominantly pass its sub-expression to the SUBSTITUTE function. The function SUBSTITUTE, on the other hand, will mainly deduce the type for concatenated expressions with prefixes. In this way, the task is divided between these two functions, again according to the D&C principle. Primitive expressions like `LiteralExpr` can deduce the type directly from their data fields. For example, the literal "`3`" can be deduced to an integer type, while the literal "`3.14`" can be deduced to a floating-point type.

Let us now return to the example of the `BinaryExpr` class. For this class, DEDUCE must find a suitable type, or rather the type of the operation result. For instance, `1+1.5` should result in `2.5`, so the deduced type must be a floating-point although the left-hand side expression is an integer. If the sub-expressions cannot be combined, as with two different types of structures, DEDUCE must terminate the type deduction process with an error. The DEDUCE function should also be provided with a parameter that specifies an optional *expected type*. This is only required for initializer expressions (i.e. the `InitializerExpr` AST class), but the interface must be uniform throughout all function implementations. Initializer expressions can only be used to initialize a variable at its declaration and its type depends on the type of that variable. An example of various initializer expressions with different types are illustrated in Listing 5.7.

Listing 5.7: Initializer Expressions in HLSL

```
1 int2                  a0    = { 1, 2 }; // 2D-Vector
2 int                   a1[2] = { 1, 2 }; // 1D-Array with 2 elements
3 struct { int x, y; } a2    = { 1, 2 }; // Anonymous structure with 2 members
```

All three initializers in this example are exactly the same. However, their types are quite different. Therefore, to deduce their types the expected type of the respective variable must be provided. DEDUCE then needs to check whether the types of the initializer elements are compatible with the expected type.

Most of the work for the function SUBSTITUTE is in the prefix expressions, i.e. the concatenated expressions from `ObjectExpr` and `ArrayExpr`. For example, the BASE type denoter needs to implement this function which is responsible for the type deduction of vector suffixes such as `vec4(1).xy` → 2D-vector. As another example, the STRUCT type denoter needs to implement this function which is responsible for the type deduction of structure members. This can be done by fetching the member via the identifier from the structure AST node, and then continue the type deduction with the fetched AST object using the DEDUCE function again. In this way, our two main functions for type deduction work in cooperation. In the case of the variable `a2` from Listing 5.7, this type deduction works as follows: `a2.x` $\xrightarrow{\text{SUBSTITUTE}}$ `int x` $\xrightarrow{\text{DEDUCE}}$ INTEGER. The function SUBSTITUTE calls FETCH on its structure AST node (i.e. `StructDecl` class) to retrieve the symbol for the variable declaration `x` (i.e. `VarDecl` class). Then the function DEDUCE is applied on this AST symbol to deduce its integer type. For long expressions, like the one in Listing 5.1 from above, the type deduction may take a considerable amount of computation time. Especially the memory access between all the references (or rather pointers) is time-consuming, due to lots of cache misses. This is why it is important to buffer the deduced type, so that further type deductions of the same expression become trivial.

Intrinsic Matching

The last major part of the type deduction we want to discuss is the type matching of intrinsics. For this, it is first of all important to know that almost all intrinsics are generically declared. This means all generic intrinsics are heavily overloaded for lots of different types. For example, the intrinsic for the trigonometric function `sin` in HLSL is declared as $\sin\colon \mathbb{R}^{M\times N} \to \mathbb{R}^{M\times N}$ with $M,N \in \{1,2,3,4\}$. As a result, the `sin` function is overloaded for scalars, vectors, and matrices of all supported dimensions (but only for floating-points, for which $\mathbb{R}$ is representative). What most shader compilers usually do to check the type compatibility and determine which overloaded function is used is that they declare all

overloaded variants of all intrinsics inductively. These declarations are commonly stored in a string which is then passed to the parser as a pre-defined part of the code. This is the simplest solution but very robust. However, it is also very inefficient since generic functions can also be analyzed generically. Providing the intrinsics as part of the code expands the original code enormously. Consequently, the parsing process becomes much slower, the AST becomes much larger, and a lot of additional functions must be handled during the type deduction. In addition, some typing errors may occur during the declaration, although the inductively approach is generally quite robust.

The approach proposed in the following is more generic, which fits well with the equally generic nature of the intrinsics. Instead of declaring all overloaded variants of the intrinsics, we declare a list of all intrinsics that is enumerated by their number of arguments. Furthermore, this list will not be declared as a string but as structure entries which are indexable in constant time. The elements of this structure contain the information of the generic return type and parameter types. In case of the `sin` intrinsic from above, we need one single structure because all overloaded variants of this function always have the same number of parameters. The structure for this intrinsic then needs an entry to describe that the return type is the same as the type of the first argument. There are also intrinsics where the return type has the same dimension as the input argument but a different scalar type. One of them is the `isinf` intrinsic which returns either `true` or `false` whether the respective vector component is infinite or finite. A list of all required generic types as well as static types is provided in the following table:

Type	Remarks
VOID	No return type.
BOOLN	Static Boolean scalar, 2D-, 3D-, or 4D-vector type where $N \in \{1,2,3,4\}$.
INTN	Static integer scalar, 2D-, 3D-, or 4D-vector type where $N \in \{1,2,3,4\}$.
UINTN	Static unsigned integer scalar, 2D-, 3D-, or 4D-vector type where $N \in \{1,2,3,4\}$.
FLOATN	Static floating-point scalar, 2D-, 3D-, or 4D-vector type where $N \in \{1,2,3,4\}$.
DOUBLEN	Static double-precision floating-point [...] type where $N \in \{1,2,3,4\}$.
GENERIC$_i$	Same type as the i-th argument.
BOOLGENERIC$_i$	Generic Boolean type with same dimension as the i-th argument.
FLOATGENERIC$_i$	Generic floating-point type with same dimension as the i-th argument.

Most parameters will be declared with the GENERIC$_i$ type name, some other parameters have static types, and a few exceptions have a static scalar type but with generic dimension. When a function call is analyzed, the compiler first tries to map the function identifier to an entry in the enumeration of intrinsics. If that fails, the compiler tries to fetch the function name from the symbol table, i.e. for non-intrinsics. If it succeeds, however, the compiler uses the structure from our intrinsic declaration list to match the argument types with the respective parameters. In addition, we also need to specify which parameters are *output parameters*. For example, the `sincos` intrinsic, which we have already mentioned in section 4.2.6, has two of them: one to assign the sine of the input argument, and another one to assign the cosine of the input argument. Otherwise, our compiler could not determine if an intrinsic parameter only allows R-value[1] arguments.

There are a few intrinsics that need a custom procedure for type analysis. One of them is the `mul` intrinsic (for matrix/vector multiplications) where the return type cannot be trivially determined by a simple declaration list. This is because the dimension of the return type depends on the dimensions of the two parameters. For instance, let be $A \in \mathbb{R}^{N_1 \times N_2}$ and $B \in \mathbb{R}^{N_2 \times N_3}$, then `mul(A, B)` $:= A \times B \in \mathbb{R}^{N_1 \times N_3}$. Hence, we need to deduce the dimension of the return type by the number of rows of the left hand side N_1 and the number of columns of the right hand side N_3. If we were to declare all combinations of this

[1] *R-value* denotes expressions that must only appear on the *right* hand side of an assignment. However, output parameters require *L-value* expressions because they will eventually appear on the *left* hand side of an assignment.

intrinsic we would end up with a very long list. A similar situation occurs with the `transpose` intrinsic where the number of rows and columns in the output matrix type are swapped.

5.2 Middle-End

5.2.1 Transformation

In the transformation phase, the AST gets converted to make the code generation a lot easier for the respective target language. This refers to the implementation of the converters in the compiler pipeline of Figure 5.1. Among other things, this is where unsupported sampler states are removed, temporary variables are inserted, and calls to member functions get converted into calls to global functions.

Statement Insertion

Since the reattachment of AST nodes is absolutely trivial, we will not discuss this further. Instead, we focus on implementing the insertion of new AST nodes that belong to the class of statements. Why this is non-trivial shows the example translation in Listing 5.8 and Listing 5.9.

Listing 5.8: Critical Expression in HLSL

```
1 for (int i = 0; i < N; V[f(i)] *= 1.5) {
2     /* ... */
3 }
```

Listing 5.9: Critical Expression in GLSL

```
1 int tmp;
2 for (int i = 0; i < N; tmp = f(i), imageStore(V, tmp, imageLoad(V, tmp) * 1.5)) {
3     /* ... */
4 }
```

In Listing 5.8 the last expression in the `for`-loop is a critical expression for a translation to GLSL. The translation looks similar to the one we have already seen in Listing 4.25, assumed that `V` denotes an RW-buffer. Since the function `f` might have side effects, such as writing to another buffer, the expression `f(i)` inside the array index must not be duplicated. This is why we need to split up the last expression into two sub-expressions: The first one stores the array index in a temporary variable, and the second one performs the actual buffer access. The translation of this single expression into two sub-expressions is actually quite easy, since we can use the so-called *sequence operator* (i.e. the comma between `...f(i), imageStore...`) to concatenate several expressions. However, we also need to insert the variable declaration statement for the temporary variable `tmp`. Finding the right place to insert this statement is the key problem we are facing here. The easiest way is to generate all those temporary variables at the beginning of the scope of each function. A more convenient result, however, can be accomplished by inserting the statement right before the position where it is needed, as in Listing 5.9. This not only ensures that the lifetime of our temporary variables is kept as short as possible, but also reduces the risk of name overlaps. Recall that mobile devices have only limited support for code optimization. Therefore, keeping the lifetime of local variables short is crucial for efficiency. There are further situations where additional statements must be inserted and a simple expression split-up is inoperative. For example, the scalar expression in a *scalar-to-structure* type cast in HLSL must be duplicated for the number of structure members in GLSL. This again involves a temporary variable as in the example above.

To solve this problem we need a handler which allows us to insert statements while we are traversing the AST. The insertion must take place *before the current statement*. If we return to the example in Listing

5.9, the *current statement* is the entire `for`-loop block at the point we are traversing the last expression within this loop statement. As soon as we traverse *into* that code block (i.e. after the open brace), the insertion must take place *inside* the code block. As a result, we cannot rely on the previous visitor pattern we used in our compiler so far. Our statement handler keeps track of the current scope and also manages the statement traversal in the current scope. Furthermore, in some situations the handler needs to insert a code-block if the respective conditional- or loop statement does not have one. This is a frequently used convention by omitting the curly braces when only a single sub-statement is defined. An illustration of this situation is shown in Listing 5.10 and Listing 5.11.

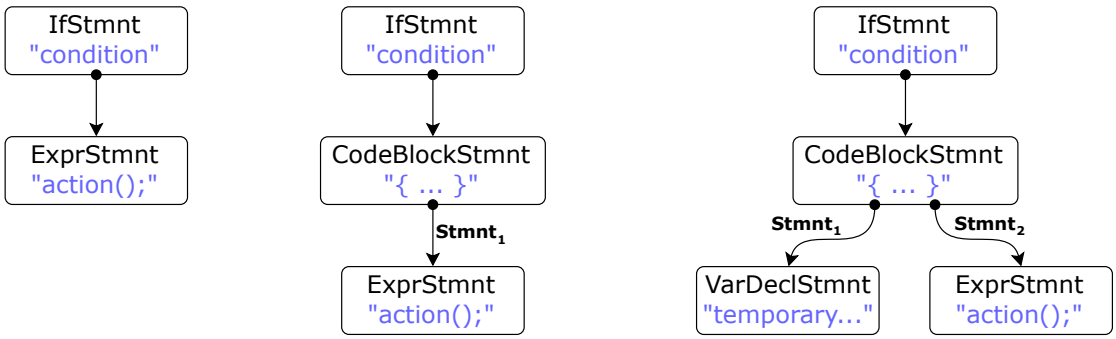
Listing 5.10: Statement without Code-Block

```
1 if (condition)
2     // <-- no further statements
3     //     can be inserted here
4     action();
```

Listing 5.11: Statement with Code-Block

```
1 if (condition) {
2     // <-- further statements
3     //     can be inserted here
4     action();
5 }
```

If a temporary variable declaration must be inserted into the conditional statement from Listing 5.10, the statement handler needs to insert a code block before additional sub-statements can be inserted. The insertion of such a code block as well as a temporary variable is illustrated in Figure 5.6 with a simplified AST. This can of course also be solved by moving this problem to the code generator. The code generator



(a) Initial AST **(b)** Code-block inserted **(c)** Temporary variable inserted

Figure 5.6: Simplified AST to illustrate the insertion of a code block into a conditional statement.

then always needs to write curly braces for conditional- and loop statements. However, our statement handler can insert these code blocks quite easily alongside the remaining transformations. This again ensures a higher correlation between input and output code.

Expression Conversion

The transformation process can be split-up according to the D&C principle, too. As we have already seen, the expressions are a complex part of our compiler, so it is reasonable to implement their conversion into a separate class and thus into a separate conversion step. This class will also assist our implementation to reduce redundant code since several expression conversions can be used for multiple target languages. Especially the conversion of implicit to explicit type casts are handled by this class. The execution of expression conversions can also be split-up into several steps. This may be required due to changes of the expressions during other conversion steps. A special characteristic of this AST visitor is the way the nodes are traversed. Consider a texture sampling call where the `Sample` HLSL intrinsic has already been converted into the respective `texture` GLSL intrinsic. If, however, the element type of the texture we are sampling from is not a 4D-vector, we also need to convert this implicit vector subscript into an explicit one. For example, such a conversion could look like this: `v=texture(...)` → `v=texture(...).rgb`. The

best location to check if such a conversion is required is inside the visitor function of the call expression. Unfortunately, a part of the conversion involves the parent AST node of that call expression, i.e. the assignment expression in this case. This is because the sub-expression of the assignment transforms from a call expression (referring to `texture(...)`) to an object expression (referring to `.rgb`), due to our prefix approach. Therefore, a member field of the parent node must be converted, too. If we stored a reference to the parent for each AST node, this would not be an issue. This in turn would make the reordering as well as the generation of AST nodes more complicated. Moreover, the expression conversion is the only task where references to the parent nodes are valuable. Hence, we implement the primary conversion check into a function which is then called for each sub-expression inside all expression visitors where they are needed. This results in a little more effort at this point, but saves us a lot of work in the remaining AST handling.

Scope Rearrangement

Every variable can only be declared once within each scope and the scope rules are almost identical between GLSL and HLSL. However, there is one difference: In GLSL, the sub-statement of `for`- and `while`-loops does not introduce a new scope for variable names. In HLSL, on the other hand, the opposite is the case. Consider a loop where the iteration variable is only required to define the number of iterations, but it is not used inside the loop body. In this case, another variable with the same identifier can be declared inside the loop. This is perfectly valid in HLSL, but results in a 'redeclaration'-error in GLSL. In consequence, we need to rename those inner variables for which we need to deduce the correct scope again. The first time we deduced the scope was during the context analysis for the source language, and now we need to deduce a new scope for the target language. Although the scope rules are only slightly different, it is advisable to deduce a full-fledged scope. This ensures a reliable and robust scope system. In addition, it allows our AST converter to detect overlapping names caused by the remaining name mangling.

5.3 Back-End

5.3.1 Code Generation

At the point we are in the back-end, our compiler already transformed the entire AST so that there is not much leftover and the final code generation can be implemented quite easily. The last analysis in the back-end is performed by the *reference analyzer*. It determines, starting from the entry point, which functions and global objects are referenced. This is important because some attributes, such as those for tessellation shaders, must not appear in other kinds of shaders. So if a single source file contains multiple shaders, which is frequently the case for HLSL, all parts of the shader that do not belong to the output entry point must be culled. After all function- and global object declarations are marked as either *referenced* or *unreferenced*, the code generator can omit all unreferenced AST nodes.

Object Referencing

The reference analyzer is one of the few AST visitors where the first node that is traversed is not the `Program` root node. As the name implies, the first node is the *entry point* which is represented as a function declaration object. Starting from this point, the function body is traversed next. Whenever a call expression appears, the declaration object that is referenced by that call will be traversed and the procedure repeats. In order to prevent the analyzer from traversing the same function more than once, it must be checked whether a function object has already been traversed or not. In this way

only the objects (and not only functions) that are actually used by the output shader will be marked as referenced. The information for the markers can be stored in a bit-mask. Further bit-fields can be added to determine which local variable is eventually written or read, which can be utilized as additional analysis and code reflection. Another feature that can be implemented in the reference analyzer alongside the object referencing is to verify whether the shader is free of recursive function calls. They are not supported in any shading language, since GPUs do not have a stack[1]. It is advantageous to perform this kind of analysis within the reference analyzer instead of the context analyzer, because the former has the adequate order of traversal. This order also allows us to track the entire call stack to output a high quality report on such recursive calls. The effect of doing this analysis in the back-end is that illegal recursive calls that do not appear in the output code are not recognized by the compiler. Which in turn is a negligible fault tolerance.

Shader Model Identification

In HLSL, the SM (also known as *target profile*) is determined by the host application when the shader is compiled. In GLSL, on the other hand, the SM or rather shader version must be specified within the shader itself. And not only that, the `#version`-directive must also be the very first statement in every GLSL shader. As a result, the required shader version must be known before any output code is generated, at least for an HLSL→GLSL translation. The easiest implementation could simply output always the highest possible shader version. However, there are many platforms that do not support the latest GLSL version. It is therefore crucial to output only the *highest necessary* but at the same time the *lowest possible* version. This is very important especially for embedded systems. We can achieve that with another traversal over the entire AST to determine which intrinsics and buffer types are used. For example, the SSBOs are only available since GLSL version 430. If none of those buffers and other languages features are used, the output version can be much lower. Note that it is beneficial to perform this process *after* the reference analyzer, since buffer objects that are not used in the output code should not affect the output version. Our compiler can also provide the option to output a fixed version and compensate missing language features by their respective GLSL extension. For instance, arrays of multiple dimensions such as `int a[4][2]` require either GLSL version 430 or the extension `GL_ARB_arrays_of_arrays`. While traversing the AST, each node can indicate which version or extension it requires and the SM analyzer can gather these information. The code generator then outputs the determined version and all required extensions.

Writing Output Code

We have finally reached the last stage of our compiler pipeline. The part that writes the output code to a stream of characters (e.g. a file stream) can be kept very simple. However, to make a couple of formatting options available to the compiler user, it is advisable to separate the actual text-writing into a distinct class which we will call the *code writer*. The interface of this class offers various basic writing functionality such as: write string of characters, begin new line, end current line, begin new scope, end current scope, increase indentation, decrease indentation. These functions can provide a flexible output formatting in collaboration. While this has only an aesthetic purpose, it can still be very useful for cross-platform developers since the coding style can vary a lot between software projects. Moreover, some programming languages even have officially specified and commonly acknowledged coding style guidelines. Violating these guidelines often leads to communication difficulties among the developers. Having a cross-compiler that provides flexible formatting can therefore be advantageous.

[1] *Stacks* are used by programming languages to store local variables and registers. This makes recursion very simple to implement on hardware level, but GPUs have a lot more registers than CPUs. Hence, providing a stack would become inefficient.

In the code generator itself, we can utilize the bidirectional keyword mappings we have discussed in section 5.1.2. With them we can convert the enumerations into their respective keywords for the target language. Here we also write all the wrapper functions whose requirements were determined during the context analysis and transformation phase. Some parts in the code generator can contain subtle problems. One of them is the generation of the *unary operator* (i.e. `UnaryExpr` AST nodes). Consider an input code with an expression like `- - x` (i.e. double negation) which results in two sub-expressions of that class. If the visitor function of our code generator simply generates a hyphen character for this subtraction operator and continues with its sub-expression, the output code would look like this: `--x`. However, this specifies only one unary expression or more specifically the *decrement operator* which is something completely different. To avoid this erroneous output, the visitor must either append a space character between the operator and the sub-expression or wrap the sub-expression in brackets.

6 Results and Discussion

We will now compare the translation results of our cross-compiler against other state-of-the-art technology. Since XShaderCompiler is just a prototype which only supports HLSL→GLSL translation at the present time, we will always use HLSL as source language and GLSL as target language. Then we discuss the results of the following translation methods: **handwritten** which is assumed to be the ground truth, **XShaderCompiler** as source-to-source approach, **glslang/SPIRV-Cross** as IR approach, and **fxc/HLSLCrossCompiler** [18] as byte-code approach in which fxc denotes the DirectX Effects Compiler to translate HLSL into byte-code. In the end, we also present a performance comparison of all applied technologies. For reasons of redundancy, we only present the results of the first example in full length. The remaining examples are shortened.

6.1 Benchmarks

6.1.1 Simple Texturing

We start with a simple texture shader as shown in Listing 6.1. It contains a very basic constant buffer to store the matrix which transforms the vertex coordinates from world-space into projection-space. The fragment shader performs a straightforward sampling function of a 2D texture.

Listing 6.1: Simple Texturing in HLSL

```
cbuffer Matrices : register(b0) {
  float4x4 wvpMatrix;
};

struct VertexIn {
  float4 position : POSITION;
  float2 texCoord : TEXCOORD;
};

struct VertexOut {
  float4 position : SV_Position;
  float2 texCoord : TEXCOORD;
};

// Vertex shader entry point
void VS(in VertexIn i, out VertexOut o) {
  o.position = mul(wvpMatrix, i.position);
  o.texCoord = i.texCoord;
}

Texture2D tex : register(t0);
SamplerState smpl : register(s0);

// Pixel shader entry point
float4 PS(in VertexOut i) : SV_Target {
  return tex.Sample(smpl, i.texCoord);
}
```

Ground Truth

In the ground truth, we keep the GLSL version as low as possible. This implies that we use OpenGL semantics instead of Vulkan semantics. Therefore, the sampler state will be removed and only the texture remains. We keep the commentaries as well as formatting style as it is. The handwritten translations are shown in Listing 6.2 and Listing 6.3.

Listing 6.2: GLSL Vertex Shader

```
#version 140

layout(std140, row_major) uniform Matrices {
  mat4 wvpMatrix;
};

in vec4 position;
in vec2 texCoord;

out vec2 vTexCoord;

// Vertex shader entry point
void main() {
  gl_Position = position * wvpMatrix;
  vTexCoord = texCoord;
}
```

Listing 6.3: GLSL Fragment Shader

```
#version 130

in vec2 vTexCoord;

out vec4 outColor;

uniform sampler2D tex;

// Pixel shader entry point
void main() {
  outColor = texture(tex, vTexCoord);
}
```

Results of XShaderCompiler

With the results of this simple texturing shader XShaderCompiler can already emphasize its benefits. The translations have exactly the same size as the ground truth, commentaries are preserved, and only a different name for the shader IO field `texCoord` is used. Thanks to the avoidance of wrapper functions as well as reference analysis, further optimization attempts are unnecessary.

Listing 6.4: GLSL Vertex Shader

```
#version 140

in vec4 position;
in vec2 texCoord;

out vec2 xsv_TEXCOORD0;

layout(std140, row_major) uniform Matrices {
  mat4 wvpMatrix;
};

//  Vertex shader entry point
void main() {
  gl_Position = (position * wvpMatrix);
  xsv_TEXCOORD0 = texCoord;
}
```

Listing 6.5: GLSL Fragment Shader

```
#version 130

in vec2 xsv_TEXCOORD0;

out vec4 SV_Target0;

uniform sampler2D tex;

//  Pixel shader entry point
void main() {
  SV_Target0 = texture(tex, xsv_TEXCOORD0);
}
```

Results of glslang/SPIRV-Cross

Listing 6.6: GLSL Vertex Shader

```
#version 450

struct VertexIn
{
    vec4 position;
    vec2 texCoord;
};

struct VertexOut
{
    vec4 position;
    vec2 texCoord;
};

struct VertexOut_1
{
    vec2 texCoord;
};

layout(binding = 0, std140) uniform Matrices
{
    layout(row_major) mat4 wvpMatrix;
} _26;

layout(location = 0) in vec4 position;
layout(location = 1) in vec2 texCoord;
layout(location = 0) out VertexOut_1 o;

void _VS(VertexIn i, out VertexOut o_1)
{
    o_1.position = i.position * _26.wvpMatrix;
    o_1.texCoord = i.texCoord;
}

void main()
{
    VertexIn i;
    i.position = position;
    i.texCoord = texCoord;
    VertexIn param = i;
    VertexOut param_1;
    _VS(param, param_1);
    VertexOut o_1 = param_1;
    gl_Position = o_1.position;
    o.texCoord = o_1.texCoord;
}
```

Listing 6.7: GLSL Fragment Shader

```
#version 450

struct VertexOut
{
    vec4 position;
    vec2 texCoord;
};

struct VertexOut_1
{
    vec2 texCoord;
};

layout(binding = 0, std140) uniform Matrices
{
    layout(row_major) mat4 wvpMatrix;
} _55;

uniform sampler2D SPIRV_Cross_Combinedtexsmpl;

layout(location = 0) in VertexOut_1 i;
layout(location = 0) out vec4 _entryPointOutput;

vec4 _PS(VertexOut i_1)
{
    return texture(SPIRV_Cross_Combinedtexsmpl, i_1.
        ↪ texCoord);
}

void main()
{
    VertexOut i_1;
    i_1.position = gl_FragCoord;
    i_1.texCoord = i.texCoord;
    VertexOut param = i_1;
    _entryPointOutput = _PS(param);
}
```

With the results of glslang/SPIRV-Cross, shown in Listing 6.6 and Listing 6.7, we have a much larger output code. The overhead comes mainly from the initialization of wrapper functions. It is therefore to be assumed that the overhead ratio will decrease with larger shaders. Nevertheless, this is a good example of the benefit of our approach compared to glslang/SPIRV-Cross. Also note that the unreferenced UBO `Matrices` has not been removed from the fragment shader in Listing 6.7.

Results of fxc/HLSLCrossCompiler

The results of fxc/HLSLCrossCompiler, shown in Listing 6.8 and Listing 6.9, are very unrelated to the input code. There are several temporary variables automatically generated by the cross-compiler. In addition, the matrix/vector multiplication has been unrolled into multiple statements. SPIR-V for instance has a distinct op-code for this operation, but in the HLSL byte code this operation is transformed into multiple operations.

Listing 6.8: GLSL Vertex Shader

```
#version 430
struct vec1 {
    float x;
};
struct uvec1 {
    uint x;
};
struct ivec1 {
    int x;
};
out gl_PerVertex {
vec4 gl_Position;
float gl_PointSize;
float gl_ClipDistance[];};
layout(location = 0) uniform struct MatricesVS_Type {
    mat4 wvpMatrix;
} MatricesVS;
layout(location = 0)  in  vec4 dcl_Input0;
vec4 Input0;
layout(location = 1)  in  vec4 dcl_Input1;
vec4 Input1;
#undef Output0
#define Output0 phase0_Output0
vec4 phase0_Output0;
layout(location = 1)  out  vec4 VtxGeoOutput1;
#define Output1 VtxGeoOutput1
vec4 Temp[1];
ivec4 Temp_int[1];
uvec4 Temp_uint[1];
void main()
{
    Input0 = dcl_Input0;
    Input1 = dcl_Input1;
    Temp[0] = Input0.yyyy * MatricesVS.wvpMatrix[1];
    Temp[0] = MatricesVS.wvpMatrix[0] * Input0.xxxx + Temp
        ↪ [0];
    Temp[0] = MatricesVS.wvpMatrix[2] * Input0.zzzz + Temp
        ↪ [0];
    Output0 = MatricesVS.wvpMatrix[3] * Input0.wwww + Temp
        ↪ [0];
    Output1.xy = Input1.xy;
    gl_Position = vec4(phase0_Output0);
    return;
}
```

Listing 6.9: GLSL Fragment Shader

```
#version 430
struct vec1 {
    float x;
};
struct uvec1 {
    uint x;
};
struct ivec1 {
    int x;
};
layout(location = 0) uniform sampler2D tex;
layout(location = 1)  in  vec4 VtxGeoOutput1;
vec4 Input1;
layout(location = 0) out  vec4 PixOutput0;
#define Output0 PixOutput0
void main()
{
    Input1 = VtxGeoOutput1;
    Output0 = texture(tex, Input1.xy);
    return;
}
```

6.1.2 Compute Shader for Filtering

The next shader we consider is a compute shader for filtering and is shown in Listing 6.10. It is a slightly modified version of a shader from the DirectX Software Development Kit (SDK) samples. The shader makes use of different language features: RW buffers, shared resources, memory barriers for synchronization between the shader invocations, texture load intrinsics, and static compile options by using macros.

Listing 6.10: Compute Shader in HLSL

```
1  Texture2D Input : register(t0);
2  RWStructuredBuffer<float> Result : register(u0);
3
4  cbuffer cbCS : register(b0) {
5    // xy: dimension of Dispatch, zw: texture size
6    uint4 g_param;
7  };
8
9  #define BLOCKSIZE_X  8
10 #define BLOCKSIZE_Y  8
11 #define GROUPTHREADS (BLOCKSIZE_X * BLOCKSIZE_Y)
12
13 groupshared float accum[GROUPTHREADS];
14
15 static const float4 Lum = float4(.299, .587, .114, 0);
16
17 [numthreads(BLOCKSIZE_X, BLOCKSIZE_Y, 1)]
18 void CS( uint3 Gid  : SV_GroupID,
19          uint3 DTid : SV_DispatchThreadID,
20          uint3 GTid : SV_GroupThreadID,
21          uint  GI   : SV_GroupIndex )
22 {
23   float4 s =
24   #ifdef CS_FULL_PIXEL_REDUCITON
25   Input.Load( uint3(DTid.xy, 0) ) +
26   Input.Load( uint3(DTid.xy + uint2(BLOCKSIZE_X*g_param.x
       ↪ ,0), 0) ) +
27   Input.Load( uint3(DTid.xy + uint2(0, BLOCKSIZE_Y*g_param
       ↪ .y), 0) ) +
28   Input.Load( uint3(DTid.xy + uint2(BLOCKSIZE_X*g_param.x,
       ↪  BLOCKSIZE_Y*g_param.y), 0) );
29   #else
30   Input.Load( uint3((float)DTid.x/81.0f*g_param.z, (float)
       ↪ DTid.y/81.0f*g_param.w, 0) );
31   #endif
32
33   accum[GI] = dot(s, Lum);
34
35   // Parallel reduction algorithm follows
36   GroupMemoryBarrierWithGroupSync();
37   if (GI < 8)
38     accum[GI] += accum[8+GI];
39
40   GroupMemoryBarrierWithGroupSync();
41   if (GI < 4)
42     accum[GI] += accum[4+GI];
43
44   GroupMemoryBarrierWithGroupSync();
45   if (GI < 2)
46     accum[GI] += accum[2+GI];
47
48   GroupMemoryBarrierWithGroupSync();
49   if (GI < 1)
50     accum[GI] += accum[1+GI];
51
52   if (GI == 0)
53     Result[Gid.y*g_param.x+Gid.x] = accum[0];
54 }
```

Ground Truth

The special thing about the translation from Listing 6.11 is that the `#ifdef`-directive in line 36 remains. This allows switching between the two render passes at compile time. Unfortunately, keeping preprocessor directives is not possible for all automatic translations that are presented here.

Listing 6.11: Compute Shader in GLSL

```
1  #version 430
2
3  uniform sampler2D Input;
4
5  layout(std430) buffer ResultBuffer {
6    float Result[];
7  };
8
9  layout(std140) uniform cbCS {
10   // xy: dimension of Dispatch, zw: texture size
11   uvec4 g_param;
12 };
13
14 #define BLOCKSIZE_X  8
15 #define BLOCKSIZE_Y  8
16 #define GROUPTHREADS (BLOCKSIZE_X * BLOCKSIZE_Y)
17
18 shared float accum[GROUPTHREADS];
19
20 vec4 Lum = vec4(.299, .587, .114, 0);
21
22 layout(local_size_x = BLOCKSIZE_X, local_size_y =
       ↪ BLOCKSIZE_Y, local_size_z = 1) in;
23
24 void GroupMemoryBarrierWithGroupSync() {
25   groupMemoryBarrier();
26   barrier();
27 }
28
29 void main() {
30   uvec3 Gid  = gl_WorkGroupID;
31   uvec3 DTid = gl_GlobalInvocationID;
32   uvec3 GTid = gl_LocalInvocationID;
33   uint  GI   = gl_LocalInvocationIndex;
34
35   vec4 s =
36   #ifdef CS_FULL_PIXEL_REDUCITON
37   texelFetch(Input, DTid.xy, 0) ) +
     ...
40   texelFetch(Input, DTid.xy + uvec2(BLOCKSIZE_X*g_param.x,
       ↪  BLOCKSIZE_Y*g_param.y), 0) );
41   #else
42   texelFetch(Input, ivec2(float(DTid.x)/81.0f*float(
       ↪ g_param.z), float(DTid.y)/81.0f*float(g_param.w)
       ↪ ), 0 );
43   #endif
44
45   accum[GI] = dot(s, Lum);
46
47   // Parallel reduction algorithm follows
48   GroupMemoryBarrierWithGroupSync();
49   if (GI < 8)
50     accum[GI] += accum[8+GI];
     ...
65   if (GI == 0)
66     Result[Gid.y*g_param.x+Gid.x] = accum[0];
67 }
```

Results of XShaderCompiler

The result of XShaderCompiler, shown in Listing 6.12, can again preserve commentaries. It also generates a wrapper function for the memory barriers, namely `GroupMemoryBarrierWithGroupSync`. In this case, all parameters of the main entry point are replaced by their respective built-in variable, although a

local variable would be more convenient. As mentioned earlier, the optional render pass that could be enabled or disabled during compile time is removed completely, due to the lack of preprocessor directive preservation.

Listing 6.12: Compute Shader in GLSL

```
#version 430

layout(local_size_x = 8, local_size_y = 8, local_size_z = 1) in;

void GroupMemoryBarrierWithGroupSync() {
  groupMemoryBarrier();
  barrier();
}

uniform sampler2D Input;

layout(std430) buffer xst_Result {
  float Result[];
};

layout(std140) uniform cbCS {
  // xy: dimension of Dispatch, zw: texture size
  uvec4 g_param;
};

shared float accum[64];

vec4 Lum = vec4(.299, .587, .114, 0);

void main() {
  uvec3 xst_temp0 = uvec3(float(gl_GlobalInvocationID.x) / 81.0f * float(g_param.z), float(gl_GlobalInvocationID.y) / 81.0f * float(g_param.w), 0);
  vec4 s = texelFetch(Input, ivec2(xst_temp0.xy), int(xst_temp0.z));
  accum[gl_LocalInvocationIndex] = dot(s, Lum);

  // Parallel reduction algorithm follows
  GroupMemoryBarrierWithGroupSync();
  if (gl_LocalInvocationIndex < 8u)
    accum[gl_LocalInvocationIndex] += accum[8u + gl_LocalInvocationIndex];
  ...
  if (gl_LocalInvocationIndex == 0u)
    Result[gl_WorkGroupID.y * g_param.x + gl_WorkGroupID.x] = accum[0];
}
```

Results of glslang/SPIRV-Cross

For the results of glslang/SPIRV-Cross, shown in Listing 6.13, we only consider the main function `_CS` that is called within the wrapper function of the entry point. We first notice that the arithmetic expression inside the load-intrinsic has been duplicated. This is allowed in this case, because this expression has no side-effects but it is a redundant computation. Although an optimizing compiler in any modern graphics driver most likely removes this redundancy, it is preferable to remove it in advance. Moreover, the literals of the floating-point constants have changed quite a bit. Their values are almost identical though. This is due to floating-point inaccuracy, but XShaderCompiler could keep the literals because they were stored as strings. This is another benefit of source-to-source compilers over IR-based cross-compilation. Also note that the set of barrier-intrinsics are different in this output. And the next result example has also a different set of barrier-intrinsics. In this case, various translations are allowed.

Listing 6.13: Compute Shader in GLSL

```
void _CS(uvec3 Gid, uvec3 DTid, uvec3 GTid, uint GI)
{
    vec4 s = texelFetch(Input, ivec3(uvec3(uint((float(DTid.x) / 81.0) * float(_34.g_param.z)), uint((float(DTid.y) / 81.0) * float(_34.g_param.w)), 0u)).xy, ivec3(uvec3(uint((float(DTid.x) / 81.0) * float(_34.g_param.z)), uint((float(DTid.y) / 81.0) * float(_34.g_param.w)), 0u)).z);
    accum[GI] = dot(s, vec4(0.29899999499320983886718750, 0.58700001239776611328125, 0.11400000005960464477539062, 0.0));
    memoryBarrierShared();
    barrier();
    if (GI < 8u)
    {
        accum[GI] += (accum[8u + GI]);
    }
    ...
    if (GI == 0u)
    {
        Result._data[(Gid.y * _34.g_param.x) + Gid.x] = accum[0];
    }
}
```

Results of fxc/HLSLCrossCompiler

For the results of fxc/HLSLCrossCompiler, shown in Listing 6.14, we only consider the main entry point. The output is once again very unrelated to the input code. There are many unnecessary bitwise con-

versions between floating-points and integrals, too. This result again has a poor maintainability, which makes it quite difficult to expand the code in the target language.

Listing 6.14: Compute Shader in GLSL

```
26 void main()
27 {
28     Temp[0].xy = vec4(gl_GlobalInvocationID.xyxx).xy;
29     Temp[0].xy = Temp[0].xy * vec2(intBitsToFloat(0x3C4A4588), intBitsToFloat(0x3C4A4588));
30     Temp[0].zw = vec4(cbCSCS.g_param.zzzw).xy;
31     Temp[0].xy = Temp[0].zw * Temp[0].xy;
32     Temp[0].xy = uintBitsToFloat(uvec4(Temp[0].xyxx).xy);
33     Temp[0].zw = vec2(intBitsToFloat(0x0), intBitsToFloat(0x0));
34     Temp[0].xyz = texelFetch(Input, floatBitsToInt(Temp[0]).xy, 0).xyz;
35     Temp[0].x = dot(Temp[0].xyz, vec3(intBitsToFloat(0x3E991687), intBitsToFloat(0x3F1645A2), intBitsToFloat(0x3DE978D5)));
36     TGSM0[int(gl_LocalInvocationIndex)].value[0x0/4u  + 0] = (floatBitsToUint(Temp[0]).x);
37     groupMemoryBarrier();
38     memoryBarrierShared();
39     Temp_uint[1] = uvec4(lessThan(uvec4(gl_LocalInvocationIndex), uvec4(8u, 4u, 2u, 1u))) * 0xFFFFFFFFu;
40     if((Temp_uint[1].x)!=0u){
41         Temp[0].y = intBitsToFloat(int(gl_LocalInvocationIndex) + 0x8);
42         Temp[0].y = uintBitsToFloat(TGSM0[floatBitsToInt(Temp[0]).y].value[(0u >> 2u) + 0]);
43         Temp[0].x = Temp[0].y + Temp[0].x;
44         TGSM0[int(gl_LocalInvocationIndex)].value[0x0/4u  + 0] = (floatBitsToUint(Temp[0]).x);
45     //ENDIF
46     }
    ...
76     if((gl_LocalInvocationIndex)==0u){
77         Temp[0].x = intBitsToFloat(int(gl_LocalInvocationID.y) * int(cbCSCS.g_param.x) + int(gl_LocalInvocationID.x));
78         Temp[0].y = uintBitsToFloat(TGSM0[0x0].value[(0u >> 2u) + 0]);
79         Result[floatBitsToInt(Temp[0]).x] = (Temp[0].y);
80     //ENDIF
81     }
82     return;
83 }
```

6.1.3 Practical Shaders

Two more examples of practical shaders have also been tested. However, we will only discuss a small excerpt of the first one, since the source codes are too large. The first example is NVIDIA FaceWorks [43]. This is a framework for realistic skin shading with a code base of roughly 1300 Lines Of Code (LOC). The second one is a shader for standard forward shading with per-pixel lighting and shadow mapping. We will return to this shader in the performance comparison in section 6.2. As last output comparison, we want to emphasize the benefits of our source-to-source approach, in which many temporary variables can be avoided. This is illustrated via the translations of a single function for diffuse lighting. The result of XShaderCompiler is shown in Listing 6.15.

Listing 6.15: Result of XShaderCompiler

```
1  vec3 EvaluateSSSDiffuseLight(vec3 normalGeom, vec3 normalShade, vec3 normalBlurred, float shadow, float curvature,
       ↪ GFSDK_FaceWorks_CBData faceworksData) {
2    // Directional light diffuse
3    vec3 rgbSSS = GFSDK_FaceWorks_EvaluateSSSDirectLight(faceworksData, normalGeom, normalShade, normalBlurred,
         ↪ g_vecDirectionalLight, curvature, g_texCurvatureLUT);
4    vec3 rgbShadow = GFSDK_FaceWorks_EvaluateSSSShadow(faceworksData, normalGeom, g_vecDirectionalLight, shadow, g_texShadowLUT);
5    vec3 rgbLightDiffuse = g_rgbDirectionalLight * rgbSSS * rgbShadow;
6
7    // IBL diffuse
8    vec3 normalAmbient0, normalAmbient1, normalAmbient2;
9    GFSDK_FaceWorks_CalculateNormalsForAmbientLight(normalShade, normalBlurred, normalAmbient0, normalAmbient1, normalAmbient2);
10   vec3 rgbAmbient0 = texture(g_texCubeDiffuse, normalAmbient0).rgb;
11   vec3 rgbAmbient1 = texture(g_texCubeDiffuse, normalAmbient1).rgb;
12   vec3 rgbAmbient2 = texture(g_texCubeDiffuse, normalAmbient2).rgb;
13   rgbLightDiffuse += GFSDK_FaceWorks_EvaluateSSSAmbientLight(rgbAmbient0, rgbAmbient1, rgbAmbient2);
14   return rgbLightDiffuse;
15 }
```

It can be seen that no temporary variables have been inserted. Also the variable declaration statements are written in the same manner as the input code. Now look at the result of glslang/SPIRV-Cross, which is shown in Listing 6.16. In this case, a lot of temporary variables are being allocated, namely `param_1` to `param_17`. Such an output is completely acceptable when the cross-compiler is used to translate the

shaders continuously at runtime, but the result is quite impractical whenever it is used for manual alteration.

Listing 6.16: Result of glslang/SPIRV-Cross

```
vec3 EvaluateSSSDiffuseLight(vec3 normalGeom, vec3 normalShade, vec3 normalBlurred, float shadow, float curvature,
    ↪ GFSDK_FaceWorks_CBData faceworksData)
{
    GFSDK_FaceWorks_CBData param = faceworksData;
    vec3 param_1 = normalGeom;
    vec3 param_2 = normalShade;
    vec3 param_3 = normalBlurred;
    vec3 param_4 = _176.g_vecDirectionalLight;
    float param_5 = curvature;
    vec3 rgbSSS = GFSDK_FaceWorks_EvaluateSSSDirectLight(param, param_1, param_2, param_3, param_4, param_5,
        ↪ SPIRV_Cross_Combinedg_texCurvatureLUTg_ssBilinearClamp);
    GFSDK_FaceWorks_CBData param_6 = faceworksData;
    vec3 param_7 = normalGeom;
    vec3 param_8 = _176.g_vecDirectionalLight;
    float param_9 = shadow;
    vec3 rgbShadow = GFSDK_FaceWorks_EvaluateSSSShadow(param_6, param_7, param_8, param_9,
        ↪ SPIRV_Cross_Combinedg_texShadowLUTg_ssBilinearClamp);
    vec3 rgbLightDiffuse = (_176.g_rgbDirectionalLight * rgbSSS) * rgbShadow;
    vec3 param_10 = normalShade;
    vec3 param_11 = normalBlurred;
    vec3 param_12;
    vec3 param_13;
    vec3 param_14;
    GFSDK_FaceWorks_CalculateNormalsForAmbientLight(param_10, param_11, param_12, param_13, param_14);
    vec3 normalAmbient0 = param_12;
    vec3 normalAmbient1 = param_13;
    vec3 normalAmbient2 = param_14;
    vec3 rgbAmbient0 = vec3(texture(SPIRV_Cross_Combinedg_texCubeDiffuseg_ssTrilinearRepeat, normalAmbient0).xyz);
    vec3 rgbAmbient1 = vec3(texture(SPIRV_Cross_Combinedg_texCubeDiffuseg_ssTrilinearRepeat, normalAmbient1).xyz);
    vec3 rgbAmbient2 = vec3(texture(SPIRV_Cross_Combinedg_texCubeDiffuseg_ssTrilinearRepeat, normalAmbient2).xyz);
    vec3 param_15 = rgbAmbient0;
    vec3 param_16 = rgbAmbient1;
    vec3 param_17 = rgbAmbient2;
    rgbLightDiffuse += GFSDK_FaceWorks_EvaluateSSSAmbientLight(param_15, param_16, param_17);
    return rgbLightDiffuse;
}
```

6.2 Performance Comparison

After we have seen a couple of output results of the different cross-compiler approaches, we can now take a look at the runtime performance results that are shown in Figure 6.1. The hardware and software setup of the testing environment is listed in the following table:

Component	Setup
Operating System (OS)	Microsoft® Windows 10 Education
CPU	Intel® Core™ i7-6700 @ 3.40 GHz
Random Access Memory (RAM)	32 GB DDR4 @ 2133 MHz
Runtime Architecture	64-bit (x64/AMD64)
C++ Compiler	Microsoft® Visual Studio Enterprise 2015

What we can notice in the benchmark results is that the glslang/SPIRV-Cross approach scales pretty well. Its increase between the lowest and highest duration is only a factor about ×1.48 at a LOC ratio of about ×48.15. The implementation of our approach needs most of the time for the parsing process which may be primarily caused by the heavy use of dynamic string objects from C++ instead of plain C strings. XShaderCompiler has still the best performance results though. Nonetheless, the performance results of the glslang/SPIRV-Cross approach are quite remarkable, since a full CFG and an IR is generated here. The byte-code approach yields the worst performance scaling. One of the reasons for this might be that it is more difficult to map Direct3D byte-code to GLSL than it is the case with SPIR-V.

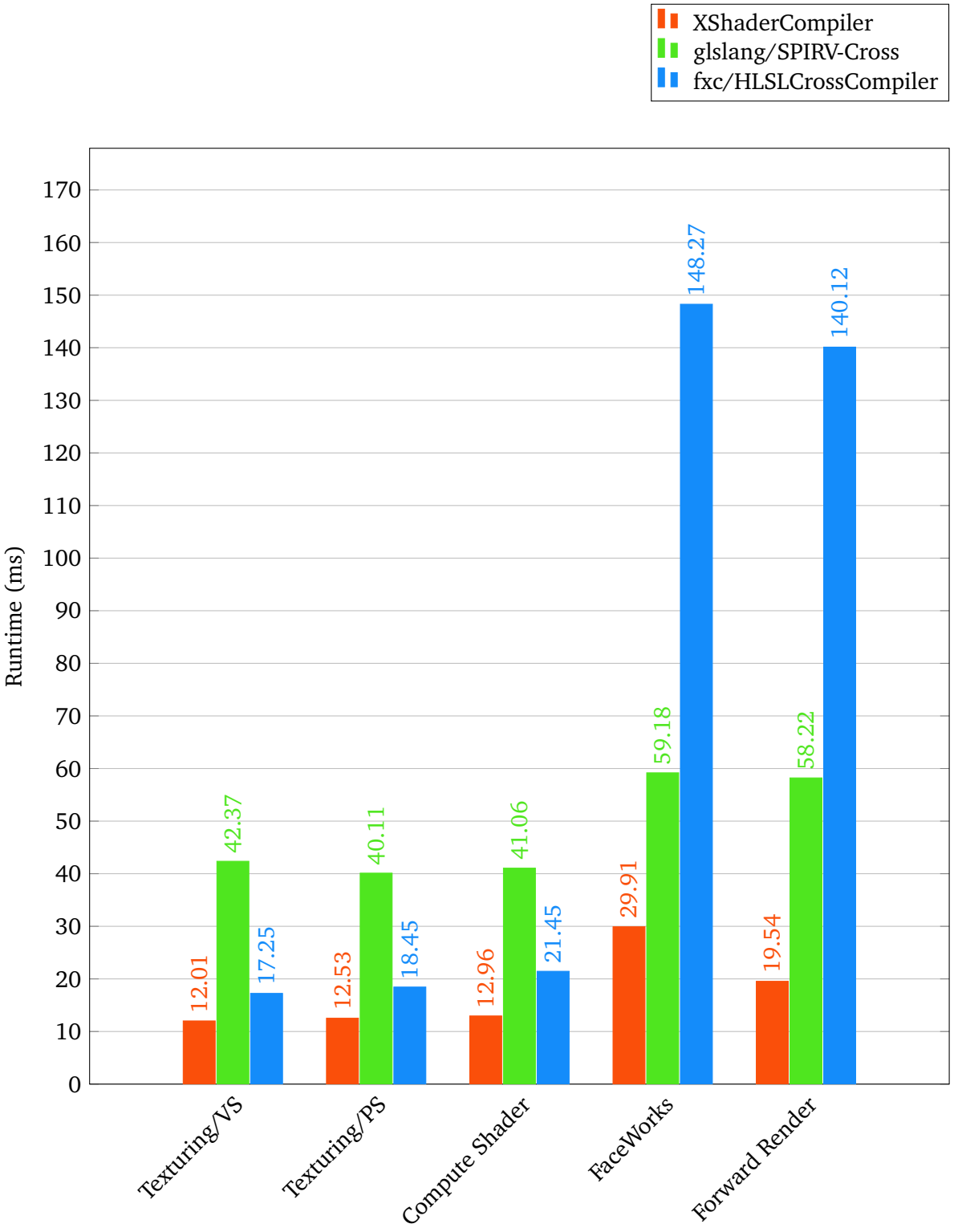


Figure 6.1: Runtime performances of the previous compilation approaches.

7 Conclusion and Future Work

7.1 Conclusion

Throughout this thesis, we have seen various approaches for automatic translations between different high-level shading languages. Besides simple approaches such as macro expansion we have mainly focused on state-of-the-art techniques. At the very beginning of this thesis, we have also shown that automatic translations for shading languages are becoming more important with the increasing appearance of mobile devices to assure a truly platform-independent infrastructure for graphics development.

The benefits of the presented technique based on the source-to-source compiler approach are clearly visible in the results we have seen in the previous chapter. Our approach is especially advantageous for shader code that is meant to be cross-compiled permanently to be used on a new rendering system. The code remains almost identical wherever possible. Temporary variables are kept to a minimum, expressions are conserved, and the formatting style is flexible. Also commentary preservation is a serious issue for large and complex shaders that are being modified and/or extended by a programmer. Moreover, our implementation achieved the best performance results in the comparison.

Of course there are some disadvantages, too. The major drawback in this approach is the poor scalability. Using an IR as a common denominator is a clear advantage over the direct path. Without an IR, the conversion between source and target language quickly becomes a bottleneck in relation to effort and complexity of the compiler. In addition, working only on the AST can become more difficult compared to a CFG, especially when transformations are operating on control-flow constructs such as loops.

7.2 Future Work

During the work on this thesis, the developments on the official shader compilers have progressed remarkably. glslang, the compiler being specified as the official GLSL shader validator by the Khronos Group, proceeded its HLSL front-end considerably and also updated its GLSL front-end to the latest version. At the beginning of 2017, Microsoft published its new HLSL compiler, named DXC, as open-source on `github.com`. In addition, developers at Google have joined the DXC project to enhance its interoperability between HLSL and SPIR-V. Especially in the year 2017, the cross-platform shader programming has thus clearly advanced. The primary shader compilers are all under active development and open for discussions and enhancements.

The XShaderCompiler project has many features left for future work as well. Although the compiler was designed generically in the first place, to support multiple source languages as well as multiple target languages, it currently only offers HLSL as input and GLSL as output. The GLSL parser has just started. In the end, XShaderCompiler can already be used for productive graphics applications in cross-platform environments though.

7.3 Appendix

A custom compiler enables many opportunities to add optional extensions to a shading language. One of them has been implemented in XShaderCompiler to provide a stronger type system for vector spaces. The original idea has been adopted from the paper *Let's Fix OpenGL* [46]. It provides a mechanism to restrict the assignments of variables to their vector space. For example, vertex coordinates are usually in model space and should not be assigned to the vertex shader output directly. Instead, they are multiplied by a projection matrix to transform them from model space into projection space. The compiler can assist the

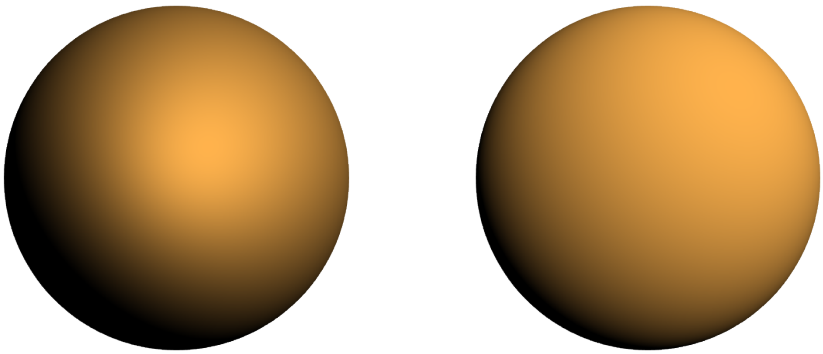
programmer by reporting an error on assignments between different vector spaces. The matrix/vector multiplication serves as change of basis. Since we developed our own compiler we can easily add an optional language extension. A proposal for such an extension is illustrated as pseudocode in Listing 7.1.

Listing 7.1: Language Extension Proposal

```
// Transform from MODEL into PROJECTION space
[space(MODEL, PROJECTION)]
float4x4 toProjection;

// Transform from MODEL into WORLD space
[space(MODEL, WORLD)]
float4x4 toWorld;

// Input position in MODEL space
[space(MODEL)]
in float4 positionIn;

// Input normal in MODEL space
[space(MODEL)]
in float4 normalIn;


// Output position in PROJECTION space
[space(PROJECTION)]
out float4 positionOut;

// Output normal in WORLD space
[space(WORLD)]
out float4 normalOut;

// OK: compiler verifies this assignment
positionOut = mul(toProjection, positionIn);

// ERROR: vector space mismatch; must be toWorld
normalOut = mul(toProjection, normalIn);
^~~~~~~~~       ^~~~~~~~~~~~
```

The extension is integrated into the attributes. For GLSL, the attribute qualifiers could be used instead. In the example, the normal vector is erroneously transformed with the projection matrix `toProjection` rather than the world matrix `toWorld`. Thanks to the stronger type system, the compiler can detect this error and could even provide a suggestion which matrix was originally meant to be used. Adopted from the output of the Clang compiler the error report could be something like: “`did you mean 'toWorld'?`”. Those type checks can be very supportive especially for shading languages where linear algebra is applied frequently.

Moreover, failures of this kind are hard to detect by just looking at the result. For example, consider the Lambertian reflectance model from Figure 2.2 again. If we were using the wrong normal vector transformation from above the result would look like shown in Figure 7.1a. The correct version, however, should look like shown in Figure 7.1b. Even an experienced graphics developer cannot always detect



(a) Wrong transformation **(b)** Correct transformation

Figure 7.1: Comparison between normal vector being transformed into *projection* space and *world* space.

those mistakes immediately. As a result, automated assistance by the compiler is highly recommendable.

Bibliography

[1] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM. doi: 10.1145/800028.808479. URL http://doi.acm.org/10.1145/800028.808479.

[2] Dana Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1):46 – 62, 1980. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(80)90041-0. URL http://www.sciencedirect.com/science/article/pii/0022000080900410.

[3] Hans-Kristian Arntzen et al. SPIRV-Cross, 2017. https://github.com/KhronosGroup/SPIRV-Cross.

[4] Valve Corporation. ToGL, March 2014. URL https://github.com/ValveSoftware/ToGL.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

[6] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321194969.

[7] Epic Games. Unreal Engine 4, 2004-2017. URL https://www.unrealengine.com/.

[8] Rich Geldreich, Dan Ginsburg, Peter Lohrmann, and Jason Mitchell. Moving Your Games to OpenGL, 2014. URL http://media.steampowered.com/apps/steamdevdays/slides/movingtoopengl.pdf.

[9] Google. ANGLE. URL https://chromium.googlesource.com/angle/angle.

[10] Ryan Gordon. MojoShader, 2008-2016. URL https://icculus.org/mojoshader/.

[11] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting Calculations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 289–298, New York, NY, USA, 1990. ACM. ISBN 0-89791-344-2. doi: 10.1145/97879.97911. URL http://doi.acm.org/10.1145/97879.97911.

[12] Evan Hart. ARB Fragment Program, 2003. URL http://www.nvidia.com/docs/IO/8228/GDC2003_OGL_ARBFragmentProgram.pdf.

[13] Anders Hejlsberg. TypeScript, 2012. URL https://www.typescriptlang.org/.

[14] Lukas Hermanns. XShaderCompiler, 2014-2017. https://github.com/LukasBanana/XShaderCompiler.

[15] Apple Inc. Metal Shading Language Specification, 2016. https://developer.apple.com/metal/metal-shading-language-specification.pdf.

[16] Gerhard Jäger and James Rogers. Formal Language Theory: Refining the Chomsky Hierarchy. 367, July 2012.

[17] Tao Jiang, Ming Li, Bala Ravikumar, and Kenneth W. Regan. Algorithms and Theory of Computation Handbook. chapter Formal Grammars and Languages, pages 20–20. Chapman & Hall/CRC, 2010. ISBN 978-1-58488-822-2. URL http://dl.acm.org/citation.cfm?id=1882757.1882777.

[18] James Jones. HLSLCrossCompiler, 2012-2016. URL https://github.com/James-Jones/HLSLCrossCompiler.

[19] John Kessenich. An Introduction to SPIR-V, 2015. URL https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf.

[20] John Kessenich. The OpenGL Shading Language, July 2017. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[21] John Kessenich, Boaz Ouriel, and Raun Krisch. SPIR-V Specification, May 2017. URL https://www.khronos.org/registry/spir-v/specs/1.2/SPIRV.pdf.

[22] John Kessenich et al. glslang, 2002-2017. https://github.com/KhronosGroup/glslang.

[23] Mark J. Kilgard. Cg in Two Pages. *CoRR*, cs.GR/0302013, 2003. URL http://arxiv.org/abs/cs.GR/0302013.

[24] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[25] Donald E. Knuth. Big Omicron and Big Omega and Big Theta. *SIGACT News*, 8(2):18–24, April 1976. ISSN 0163-5700. doi: 10.1145/1008328.1008329. URL http://doi.acm.org/10.1145/1008328.1008329.

[26] Robert Konrad. krafix - SPIR-V based GLSL cross-compiler, 2015-2017. https://github.com/Kode/krafix.

[27] Gavin S. P. Miller Kyle Hegeman, Nathan A. Carr. Particle-Based Fluid Simulation on the GPU. Computational Science - ICCS 2006, Springer, Berlin, Heidelberg, 2006. Springer. doi: 10.1007/11758549_35. URL http://link.springer.com/chapter/10.1007/11758549_35.

[28] Johann Heinrich Lambert. Photometria Sive de Mensura et Gradibus Luminus Colorum et Umbrae. 1760.

[29] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[30] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[31] Timothy Lottes. Fast Approximate Anti-Aliasing (FXAA), February 2009.

[32] Microsoft. DirectX Intermediate Language, 2017. URL https://github.com/Microsoft/DirectXShaderCompiler/blob/master/docs/DXIL.rst.

[33] Torben Ægidius Mogensen. *Basics of Compiler Design*. University of Copenhagen, anniversary edition, August 2010. ISBN 9788799315406.

[34] NVIDIA. Cg FAQ, April 2012. https://developer.nvidia.com/cg-faq.

[35] NVIDIA. GPU-Based Deep Learning Inference: A Performance and Power Analysis. NVIDIA Corporation, 2015. URL https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf.

[36] Michael Oren and Shree K. Nayar. Generalization of Lambert's Reflectance Model. In *In SIGGRAPH 94*, pages 239–246. ACM Press, 1994.

[37] Boaz Ouriel et al. SPIR 1.2 Specification for OpenCL, 2013. URL https://www.khronos.org/files/opencl-spir-12-provisional.pdf.

[38] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. ISSN 1097-024X. doi: 10.1002/spe.4380250705. URL http://dx.doi.org/10.1002/spe.4380250705.

[39] Terence Parr. ANTLR 4 Documentation. URL https://github.com/antlr/antlr4/blob/master/doc/index.md.

[40] Terence Parr and Kathleen Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993548. URL http://doi.acm.org/10.1145/1993498.1993548.

[41] Aras Pranckevičius. hlsl2glslfork, 2010-2016. URL https://github.com/aras-p/hlsl2glslfork.

[42] Dan Quinlan and Chunhua Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[43] Nathan Reed. Advanced Skin Shading With FaceWorks. NVIDIA Corporation, March 2014.

[44] Guido Rossum. Python Reference Manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[45] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, 3rd edition, 2009. ISBN 0321637631, 9780321637635.

[46] Adrian Sampson. Let's Fix OpenGL. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-032-3. doi: 10.4230/LIPIcs.SNAPL.2017.14. URL http://drops.dagstuhl.de/opus/volltexte/2017/7130.

[47] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0131387685, 9780131387683.

[48] Daniel Schlegel. Deep Machine Learning on GPUs. University of Heidelberg, ZITI, January 2015. URL http://www.ziti.uni-heidelberg.de/ziti/uploads/ce_group/seminar/2014-Daniel_Schlegel.pdf.

[49] Axel-Tobias Schreiner and James Heliotis. Design patterns in parsing, 2008. URL http://scholarworks.rit.edu/other/82.

[50] R. S. Scowen. Extended BNF - A Generic Base Standard. In *Proceedings of the 1993 Software Engineering Standards Symposium (SESS'93)*, August 1993. URL http://www.cl.cam.ac.uk/~{}mgk25/iso-14977-paper.pdf.

[51] Mikko Strandborg. HLSLcc, 2016. URL https://github.com/Unity-Technologies/HLSLcc.

[52] Bjarne Stroustrup. History of Programming languages—II. chapter A History of C++: 1979–1991, pages 699–769. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1. doi: 10.1145/234286.1057836. URL http://doi.acm.org/10.1145/234286.1057836.

[53] Adobe Systems. What is AGAL, October 2011. URL http://www.adobe.com/devnet/flashplayer/articles/what-is-agal.html.

[54] Jonathan Tompson and Kristofer Schlachter. An Introduction to the OpenCL Programming Model. *Person Education*, 49, 2012.

[55] UnknownWorlds. hlslparser, march 2014. URL https://github.com/unknownworlds/hlslparser.

Acronyms

AGAL	Adobe Graphics Assembly Language
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
CFG	Control Flow Graph
Cg	C for Graphics
CGI	Computer Generated Imagery
CPU	Central Processing Unit
CST	Concrete Syntax Tree
D&C	Divide-and-Conquer
DAST	Decorated Abstract Syntax Tree
DFA	Deterministic Finite Automaton
DRY	Don't Repeat Yourself
DXC	DirectX Shader Compiler
DXIL	DirectX Intermediate Language
DXIR	DirectX Intermediate Representation
EBNF	Extended Backus Naur Form
ESSL	OpenGL ES Shading Language
FXAA	Fast Approximate Anti-Aliasing
GLSL	OpenGL Shading Language
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HLSL	DirectX High Level Shading Language
IO	Input/Output
IR	Intermediate Representation
JIT	Just-in-Time
LBA	Linear-bounded Automaton
LLVM	Low Level Virtual Machine
LOC	Lines Of Code
LOD	Level Of Detail
MSL	Metal Shading Language
O-O	Object-Oriented
OS	Operating System
PDA	Push-down Automaton
RAM	Random Access Memory
RSL	RenderMan Shading Language
RW	Read/Write
SDK	Software Development Kit
SM	Shader Model
SPIR-V	Standard Portable Intermediate Representation
SSA	Static Single Assignment
SSBO	Shader Storage Buffer Object
TAC	Three-Address Code
TM	Turing Machine
UBO	Uniform Buffer Object
UML	Unified Modeling Language