FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Shading Framework for Modern Rendering Engines

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Onur Dogangönül
Matrikelnummer 0416109

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 06.11.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Shading Framework for Modern Rendering Engines

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Onur Dogangönül
Registration Number 0416109

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 06.11.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Onur Dogangönül
Am Raun 28, 6460 Imst

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

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I want to thank Michael Wimmer for supervising this thesis and supporting me. Also, I want to thank David Schedl for his assistance at the beginning of this thesis. Credits to Stefan Reinalter, who provided a detailed feedback on the problem definition. I am obliged to the whole institute of Computer Graphics and Algorithms at the Vienna University of Technology for giving me the possibility to learn from an excellent scientific staff.

I want to thank my friends, Helene Hovorka and Hannes Widmoser, for proof-reading parts of the thesis and helping with translation issues.

Last but not least, I want to thank my parents, Muharrem and Zeynep, for supporting me throughout my education.

Abstract

Nowadays real-time graphics programmers find themselves in a challenging development environment. Graphics algorithms are usually accelerated by specialized hardware, which is optimized for streaming operations and parallel computation. Programmers write shaders to specify the behavior of certain stages of the graphics pipeline. Shaders are mostly written in procedural per-stage languages like HLSL or GLSL, whereas application code is mostly written in C/C++. Hardware state setup, shading- and application code has to be aligned to achieve a certain visual effect. Advanced graphics algorithms often comprise numerous shader files, including undesirable duplicate- and pass-through shader code. Furthermore, code reusability, composability and modularity is restricted due to the design of current shading languages. In this thesis current issues in real-time graphics development are discussed. Particularly, the Spark framework, which is considered as a reference approach is examined and extended by supplementary documentation and functionality. Within the framework the authors present a novel aspect-oriented per-pipeline shading language, which is eventually translated to HLSL procedures by a compiler on top of a Direct3D 11 back-end. Within this thesis an OpenGL 4.2 back-end is presented, new examples are shown and the extended framework is re-evaluated and discussed.

Kurzfassung

Die Entwicklung von modernen Echtzeitgrafikanwendungen stellt für Entwickler eine große Herausforderung dar. Grafikalgorithmen werden üblicherweise von spezialisierter Hardware, die für parallele Berechnung einer großen Anzahl an Operationen optimiert ist, ausgeführt. Die Entwickler definieren das Verhalten der verschiedenen Abschnitte der Grafikpipeline über Shaderprogramme, die typischerweise in prozeduralen Sprachen wie HLSL oder GLSL für jeden Abschnitt separat geschrieben werden. Im Gegensatz dazu wird der Anwendungscode selbst meist in C/C++ geschrieben. Die Konfiguration der Hardware, der Shader- und der Anwendungscode müssen miteinander abgestimmt sein um bestimmte visuelle Effekte erzielen zu können. Die meisten komplexen Grafikalgorithmen erfordern zahlreiche Shaderdateien, die zum Teil Code doppelt enthalten oder nur dafür dienen, Daten von einem Abschnitt der Pipeline in den nächsten weiterzuleiten. Dazu kommt, dass die Wiederverwendbarkeit und Modularität des Codes durch das Design der Shadersprachen eingeschränkt ist. Diese Arbeit setzt sich mit aktuellen Themen der Entwicklung von Echtzeitgrafikanwendungen auseinander. Das Spark Framework wird dabei als Referenzansatz vorgestellt, untersucht und durch unterstützende Dokumentation und Funktionalität erweitert. Die Autoren des Spark Frameworks stellen eine neuartige, aspektorientierte Shadingsprache vor, die durch einen Compiler in HLSL Prozeduren basierend auf einem Direct3D Backend umgeschrieben wird. Im Rahmen dieser Arbeit wird ein OpenGL Backend präsentiert, durch neue Beispiele erläutert und das resultierende erweiterte Framework evaluiert und untersucht.

Contents

CHAPTER 1

Introduction

1.1 Real-Time Computer Graphics

A usual graphics application renders complex scenes, comprising hundreds of 3D models, for a given camera setup and several light sources, to one or more output devices. Location and shape of the visual objects are determined by their geometry and the projection properties of the camera [15, Chapter 2]. Their appearance, however, is specified by defining materials, textures, light sources and, of course, shading models [15, Chapter 2]. Since real-time applications require an interactive setting, at least 72 frames per second (FPS) have to be provided to guarantee continuous rendering [15, Chapter 1]. Generally, a trade-of between performance and physical authenticity has to be made. Material-light interactions and other phenomena are often merely based on physical concepts but still require strong visual plausibility. The development of graphics algorithms, shading languages and the design of hardware architectures is intrinsically coupled. In fact, dedicated graphics hardware is essential in modern real-time graphics development. A lot of efforts are made to devise optimized graphics effects to take advantage of a given hardware architecture. Conversely, new ideas in computer graphics often influence design decisions of hardware vendors.

1.2 Graphics Pipeline

An efficient implementation of the previously mentioned rendering task implicitly imposes a *pipeline structure*, with certain *stages*, which is referred to as the *graphics pipeline*. Before determining pixel colors on the screen, several distinct stages, with different tasks, have to be passed. For instance, the 3D models are usually specified in a local coordinate system. Model vertices have to be transformed from local- to projected coordinates in an earlier stage, so that in a later stage pixel colors can be determined for geometric primitives visible on the screen. Nowadays, specialized hardware accelerates most stages of the rendering pipeline, however, the

actual pipeline structure depends on the actual implementation. Usually the essential features are categorized into three conceptual stages [15, Section 2.1]:

Application stage

The *application stage* is the entry-point of the graphics application. To some extent, a so-called *rendering engine* is implemented, typically in C++, that manages and performs operations on the data structures of the scene. Further, input from possibly multiple sources have to be handled. Graphics application programming interfaces (APIs), for instance, Direct3D or OpenGL, provide means to defer graphics commands to the specialized hardware. Since the application stage is processed on the central processing unit (CPU), the developers have a lot of flexibility to perform optimizations (e.g. view frustum culling) to reduce the load of the subsequent stages.

Geometry Stage

The *geometry stage* receives all the geometric data, with according transformation matrices and other necessary resources. Since models are specified in a local coordinate system (model space), a common coordinate system (world space) is required to put all models into a scene. Therefore, a *model matrix* is associated with each primitive to provide the transformation from local to world coordinates. Subsequently, the virtual 3D scene has to be projected on a plane. To simplify projection, all visual objects are transformed into another coordinate system (view space), where the camera is located in the center (view coordinates). The transformation process from model space to view space is, usually, combined to a *model-view transform*. Depending on the implemented effects, per-vertex shading equations may be evaluated before applying projection, which is, commonly, referred to as *vertex shading*. Also, more detailed 3D-models may be generated on the fly or even geometry amplification may be implemented by optional *tessellation* and *geometry* stages. Anyhow, *projection* transforms the view volume, which is defined by a perspective or orthographic projection matrix, into a *unit cube*. Primitives inside the unit cube are send to subsequent stages, whereas primitives outside of the cube are discarded. Intersecting primitives are clipped against the unit volume. The process of *clipping* may, therefore, introduce new vertices. Moreover, vertices inside the unit cube are transformed from, so-called, *normalized device coordinates* to *screen coordinates* by the process of *screen mapping*.

Rasterisation Stage

The *rasterisation stage* receives the projected vertices and necessary data and as a first step supplementary data is computed per triangle. This process is called *triangle setup*. Subsequently, *scan conversion* (also known as *triangle traversal*) generates *fragments* for each pixel that is covered by the triangle. Vertex attributes have to be interpolated among triangle vertices to allow *per-fragment shading* operations. In a final step the generated fragment color is *merged* with the pixel information, which is stored in the framebuffer.

Additional details about these conceptual stages will be provided throughout the thesis as necessary. Stages are processed in parallel and the performance of the whole pipeline depends on its slowest stage, since other stages are dependent [15, Section 2.1]. Note that these conceptual

stages may also deploy a pipeline with sub-stages [15, Section 2.1]. A nonpipelined system that is divided into n stages, ideally benefits of a speedup by the factor of n [15, Section 2.1].

1.3 Graphics Hardware and Parallelism

Graphics applications mostly require two kinds of specialization [10, Chapter 12]:

- Streaming operations: reading a lot of data from single or sequential locations
- Parallel computation: perform independent calculations for a collection of elements of the same kind: e.g.
  - Transform vertices from model space to view space
  - Apply per-fragment shading equations

Foley distinguishes the concepts of *data parallelism* and *pipeline parallelism* [3, Section 1.1]. *Data parallelism* is accomplished by, for instance, processing all vertex computations of a primitive- or evaluating shading equations for several pixels at once. *Pipeline parallelism* addresses the fact that, for example while per-fragment calculations are performed for the current model, per-vertex computations may already be performed for another model. Since modern central processing units (CPUs) favor hardware architectures for a broader field of application and do not exhibit such levels of parallelism, graphics pipelines are implemented as special hardware circuits optimized for graphics processing. Recent trends towards general-purpose computing on graphics processing units (GPGPU) augment the processing flexibility of graphics hardware and introduce novel possibilities for other scientific fields. A few notes about GPGPU and real-time rendering are provided in Section 1.4. Nevertheless, current real-time rendering mostly involves the development of *shaders*, which are programs that specify the functionality of certain stages in the graphics pipeline.

1.4 Shader Programming

Programmable graphics pipelines were established with the introduction of NVIDIA's GeForce3 on consumer level in 2001 [15, Section 3.3]. At that time, the previously configurable graphics hardware was enhanced by programmable *vertex shaders* [15, Section 3.3]. Early graphic APIs (Microsoft's DirectX 8.0 or OpenGL via extensions) exposed new hardware features to graphics developers by an assembly-like language [15, Section 3.3]. Graphics acceleration chips developed rapidly and high-level languages, for example Microsoft's High-Level Shading Language (HLSL), OpenGL Shading Language (GLSL) or Nvidia's C for Graphics (Cg), were introduced. A shader is a C-like program that receives a set of input data and performs computations at a given rate, for instance, per-vertex or per-fragment. The outputs of a shader can be used as inputs to subsequent stages. Having discussed the coarse structure of the rendering pipeline and the basics of shader programming, it is time to take a closer look at a reference implementation. As previously mentioned, the conceptual geometry- and rasterizer stages are, nowadays, accelerated by graphics hardware. Therefore, programmers have to pass all necessary data to the

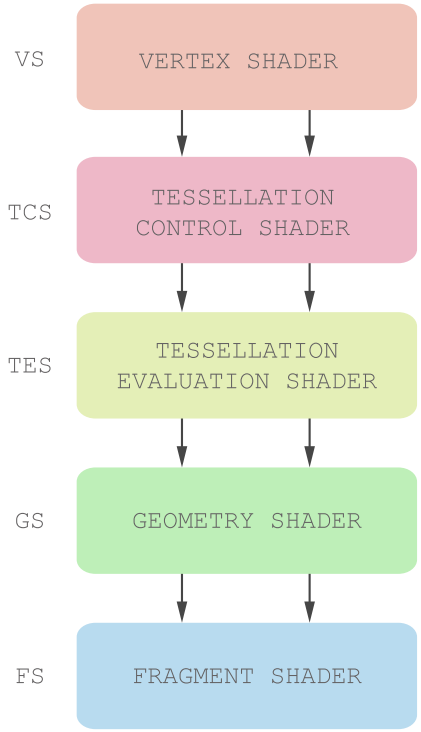

Figure 1.1: Programmable stages of the OpenGL 4.2 pipeline

graphics card and specify all *configurable* and *programmable* stages. Figure1.1 shows the decisive programmable stages of the current OpenGL 4.2 pipeline [11] [7]. Common development tasks are delineated for each stage in the next subsections.

Vertex Shader

Typically, a scene contains various 3D-models, each comprising at least several vertices specified in a local or global coordinate system. Optionally, other *vertex attributes*, for instance vertex normals, texture coordinates or material properties are provided. The vertex shader, usually, performs model-view transform, vertex shading and the projection of vertices. Each vertex shader is constrained to perform *local* computations [7, Section 2.1], which facilitates parallel computing.

Tessellation Shader

Modern graphics hardware allows on-the-fly tessellation of coarse models. The benefits [10, Chapter 13] are, for instance, physics calculations can be performed on the coarse model, while a tessellated model is rendered on screen. Further, less data (only the coarse geometry) is sent to the graphics card, since the tessellated model is re-created each frame. Also, dynamic level

of detail (LOD) algorithms may be implemented on the GPU. In OpenGL 4.2 two shaders have to be specified to perform tessellation:

- The *tessellation control* shader [7, Section 2.2] receives a patch of incoming vertices and emits a new output patch. The tessellation control shader is invoked for each vertex in the output patch. Additionally, the tessellation primitive generator (TPG), which implements the actual tessellation process, has to be configured. Inner and outer tessellation levels are specified to control the tessellation granularity.
- The *tessellation evaluation* shader [7, Section 2.3] is processed after the TPG and is invoked for each generated parametric location to interpolate actual fine vertex coordinates- and attributes.

The tessellation stage is optional. Usually, when tessellation is enabled algorithms work on control-points, which are used to interpolate a curve or a surface. Therefore, the role of the vertex shader changes. Projection is usually deferred to later stages and the vertex shader becomes a vertex shader of control-points [10, Section 13.1.1].

Geometry Shader

The geometry shader is also optional and is invoked once for each primitive and can output zero, one or more primitives [16, Chapter 6]. Input and output primitive types may diverge but only a single kind of output primitive may be specified [16, Chapter 6].

Fragment Shader

The fragment shader is executes for each fragment, which is generated by the fixed-function scan conversion stage. Pixel colors are generated and merged with the current framebuffer.

Configurable stages

Beside the programmable stages, some tasks, for instance, clipping, screen mapping, scan conversion or output merging are either fixed-operation or merely configurable. In this thesis, only the decisive programmable stages are focused and details are revealed as necessary.

Modern Pipeline Stages and Group-wise Operations

For the modern graphics pipeline *point-wise* and *group-wise* shading code may be distinguished [3, Section 1.3]. Early rendering pipelines only supported vertex and fragment shaders. The idea was to apply a custom point-wise operation over a stream of vertices and fragments, respectively. Since there is no data dependency, parallel computation is simple. However, modern stages involve group-wise operations. The tessellation control shader is invoked for each vertex in the output patch and has access to all per-vertex data in the input patch. Additional per-vertex attributes may be computed for the output patch, which is passed along to the tessellation evaluation shader. The tessellation evaluation shader is invoked for each generated parametric location.

Usually, per-vertex information is required from the output patch to compute bilinearly interpolated actual fine vertex locations. The geometry shader works on an aggregate of vertices, which form a primitive, and may even output multiple primitives.

Compute Interfaces and GPGPU

Due to the increasing computing power and flexibility of the graphics hardware, there is a growing interest to deploy the GPU for a broader field of application other than real-time rendering. GPGPU and compute approaches were initially implemented by abstracting over rendering architectures like OpenGL [3, Section 2.2.2]. CUDA or OpenCL do not conform to such abstractions and expose additional capabilities [3, Section 2.2.2]. Most prominently, Direct3D 11 and OpenGL 4.3 introduce an additional *compute shader*, which is not part of the direct rendering pipeline, but can read and write to GPU resources [8, Section 2.6] [10, Chapter 12]. Furthermore, compute shader outputs may be bound as resources to the rendering pipeline, which conveniently increases the flexibility of graphics hardware [10, Chapter 12].

OpenGL and Direct3D:

The thesis is written with a focus on OpenGL, however, also Direct3D 11 examples may be found since OpenGL 4.2/4.3 and Direct3D 11 expose similar capabilities and programmable shaders. The introduction to real-time rendering and the graphics pipeline also applies (with minor changes) to Direct3D 11. Having discussed the basic aspects of real-time rendering and shader programming, current development issues are depicted in the following section.

1.5 Problem Statement

Real-time graphics programmers face a challenging development environment. Due to the heterogeneousness of the development tools, it is not simple to apply appropriate software design decisions. Usually, C++ application code, hardware state setup and procedural shader code has to be aligned to implement graphics effects.

Figure 1.2 shows a vertex and fragment shader in GLSL. Basically, three features are implemented:

- Red highlights indicate the transformation from model coordinates to projected vertices, which is mandatory for most shading applications.
- Green highlights show lines of code corresponding to texturing.
- Blue highlights show an implementation of a simple diffuse lighting model.

Foley discusses ubiquitous drawbacks in current shader development in [3, Section 1.3] of his PhD thesis, which may also be observed in this simple application.

```
#version 420 core

uniform Uniforms
{
     mat4 modelView;
     mat4 proj;
     mat4 view;
     mat3 normalMatrix;
     vec4 w_lightPos;

} uniforms;

in vec3 vertex;
in vec3 normal;
in vec2 texCoord;

out VS2PS
{
     vec3 v_normal;
     vec3 v_lightDir;
     vec2 texCoord;

} vs2ps;

void main(void)
{
     vec4 pos = vec4(vertex, 1.0f);
     vec4 v_pos = uniforms.modelView * pos;

     vec4 v_lightPos = uniforms.view * uniforms.w_lightPos;
     vec3 v_lightPos3 = v_lightPos.xyz / v_lightPos.w;

     vec3 v_normal = uniforms.normalMatrix * normal;

     vec3 v_pos3 = v_pos.xyz / v_pos.w;
     vec3 v_lightDir = normalize(v_lightPos3 - v_pos3);

     vs2ps.v_normal = v_normal;
     vs2ps.v_lightDir = v_lightDir;
     vs2ps.texCoord = texCoord;

     gl_Position = uniforms.proj * v_pos;
}
```

```
#version 420 core

uniform sampler2D tex;

in VS2PS
{
     vec3 v_normal;
     vec3 v_lightDir;
     vec2 texCoord;

} vs2ps;

void main(void)
{
     vec4 diffuse = texture(tex, vs2ps.texCoord);
     float lighting = max(0.0f,
          dot(normalize(vs2ps.v_lightDir),
          normalize(vs2ps.v_normal)));
     vec4 color = diffuse * lighting;

     gl_FragData[0] = color;
}
```

Figure 1.2: Basic GLSL vertex and fragment shader

Lack of Modularity

Due to the procedural programming paradigm, developers may combine simple procedures to build complex procedures. Almost any kind of operations may be performed. From data dependent flow control to program loops, current shading languages are flexible enough to allow the implementation of a variety of graphics effects. However, procedural programming exposes limited structural properties. In particular, the developer is constrained to break up the graphics algorithms into procedural shaders. Again, Figure1.2 shows three different modules that are mixed together and spread over two stages. Ideally, different features should be localized and implemented in independent modules. For instance, the simple diffuse lighting model could be implemented in a separate module to be reused in other applications. Also shader extension and combination is often related to duplicate code.

Lack of Reusability

As already indicated, reusability is restricted. Developers are often tempted to utilize 'copy-paste' programming. For example, some kind of texturing code is used in most shading tasks. Usually, the code base is (to a certain degree) polluted with duplicate shader code. Changes to a particular feature require a lot of effort to be adapted to the whole project.

Additional Plumbing Code

In Figure 1.2, texture coordinates have to be passed along from the vertex shader to the fragment shader (green highlights). Although texturing is applied in the fragment shader, there is a dependency in the vertex shader.

1.6 Motivation and Aim

Aim of this Thesis

Procedural shading languages merely abstract the lowest hardware levels and were designed to allow a wide range of computations. In this thesis, the initial question is whether it is possible to provide a flexible high performance shading framework which solves the development issues illustrated in Section 1.5. The framework should cover most features of the current graphics pipeline (OpenGL 4.2 or Direct3D 11). Furthermore, and most importantly, additional run-time costs should be tolerable. This thesis examines, documents and extends an existing approach in current graphics research, which is mainly discussed in Foley's PhD thesis [14]. There, the authors present a flexible novel per-pipeline shading language (Spark), which facilitates current shader development and is therefore considered as the central subject and resource of this thesis. The aim of this thesis is to describe the absent parts of the available documentation and to provide a low-level view on the Spark compiler. Additionally, the Spark system should be extended by an OpenGL 4.2 back-end.

The Spark Framework

Given the increasing complexity of graphics algorithms, the authors of Spark [14] show the importance of adopting appropriate software-engineering principles to the difficulties of modern shader development. A main focus is the *separation of concerns*: the factoring of logically distinct program features into localized and independent modules [14]. Again, as pointed out in Section 1.5, ideally, different program features should be expressed as *separate, reusable modules*. Data which is used in later stages should be *automatically plumbed* through intermediate stages.

Figure1.3 shows a possible implementation of the simple shading application from 1.2 using Spark. Without going into language details at this point, notice that the program features are separated in *classes*, which may be reused or extended by *inheritance*. A more realistic and elaborate graphics application, where an animated, tessellated and displaced model is rendered to all six faces of the cube map in the geometry shader may be found in Foley's article [14].

```
abstract shader class Base extends OpenGL42DrawPass
{
     // @Uniform
     input @Uniform mat4 world;
     input @Uniform mat4 view;
     input @Uniform mat4 proj;

     @Uniform mat4 modelView = view * world;
     @Uniform mat4 mvp = proj * modelView;
     @Uniform mat3 normalMatrix =
          mat3(transpose(inverse(modelView)));
     // @AssembledVertex
     input @AssembledVertex vec3 pos;

     // @RasterVertex
    override RS_Position = mvp * vec4(pos, 1.0f);

     // @Pixel
     abstract output @Pixel vec4 target;
}
```

```
shader class Texturing extends Diffuse
{
     // @Uniform
     input @Uniform sampler2D tex;

     // @AssembledVertex
     input @AssembledVertex vec2 texCoord;

     // @Fragment
     override diffuse = texture(tex, texCoord);
}
```

```
shader class Diffuse extends Base
{
     // @Uniform
     input @Uniform vec4 w_lightPos;

     // @AssembledVertex
     input @AssembledVertex vec3 normal;

     // @CoarseVertex
     @CoarseVertex vec4 v_pos = modelView * vec4(pos, 1.0f);
     @CoarseVertex vec3 v_pos3 = v_pos.xyz / v_pos.w;

     @CoarseVertex vec4 v_lightPos = view * w_lightPos;
     @CoarseVertex vec3 v_lightPos3 = v_lightPos.xyz / v_lightPos.w;

     @CoarseVertex vec3 v_lightDir = normalize(v_lightPos3 - v_pos3);
     @CoarseVertex vec3 v_normal = normalMatrix * normal;

     // @Fragment
     virtual @Fragment vec4 diffuse = vec4(1.0f, 1.0f, 1.0f, 1.0f);
     @Fragment float lighting =
          max(0.0f, dot(normalize(v_lightDir), normalize(v_normal)));
     @Fragment vec4 color = diffuse * lighting;

    // @Pixel
    override target = color;
}
```

Figure 1.3: Base Spark Module

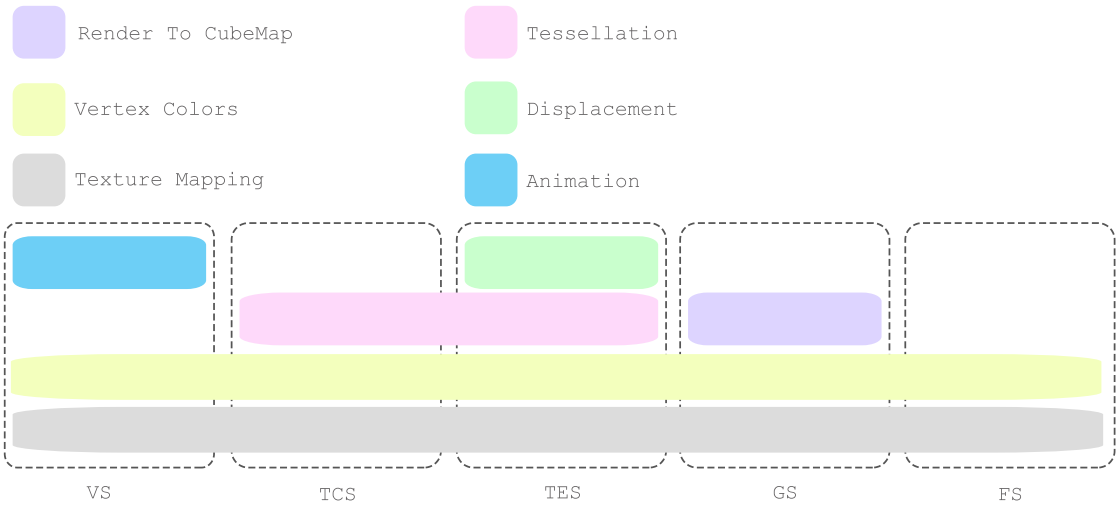

Figure 1.4: Complex visual effect with cross-cutting modules

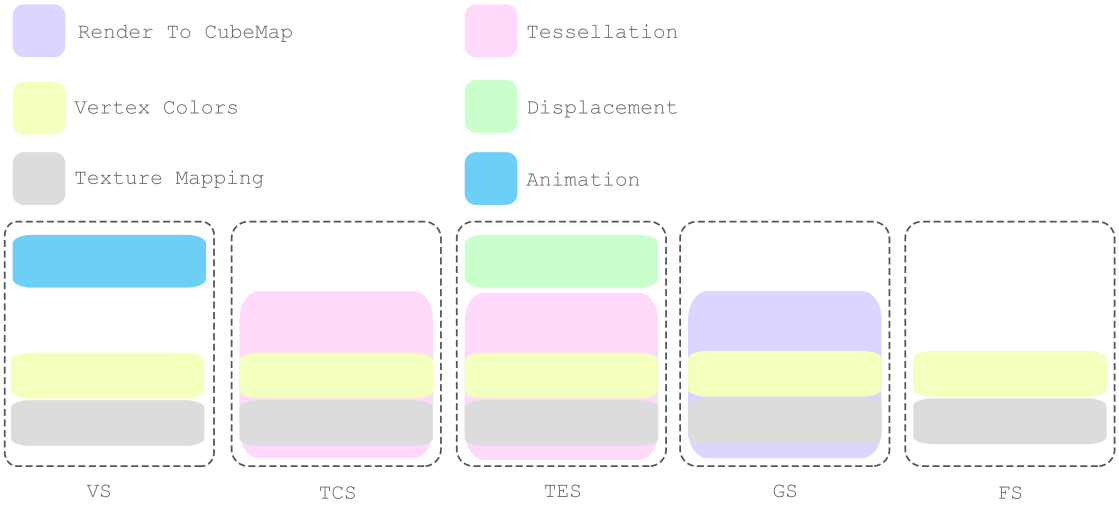

Figure 1.5: Forced decomposition of concerns, due to per-stage shading language

Figure1.4 presents an OpenGL 4.2 adaption of the same example. Dashed boxes indicate the programmable stages of the current OpenGL 4.2 pipeline. Colorized rounded rectangles denote different program features as modules. Some of them extend over multiple stages and might be considered as *cross-cutting concerns* in terms of aspect-oriented software development [14].

Figure1.5 visualizes the constrained mapping of program features to the programmable stages when utilizing a per-stage language like GLSL. Some concerns are divided upon stages, others have to be merged. Usually, the implementation of a complex effect takes some time.

Ideally, the code is cleanly separated in modules, which might be re-used when developing new effects. Given the current progress in graphics development, the authors of Spark are convinced that the time is right to re-evaluate the design criteria for real-time shading languages [14]. Spark is built on top of a Direct3D 11 back-end: Spark shaders are eventually compiled to HLSL blobs. Evaluations indicate that Spark shaders perform within 2% of optimized analog HLSL Über-shaders [14]. In conclusion of this section, it should be clarified again that the examples and figures in this section are derived from examples in [3, Section 1.4] and [14], with the intention to be adopted to the OpenGL context.

1.7 Methodical Approach

Currently, there are three online resources documenting the high-level concepts of Spark:

- ACM TOG paper on Spark (SIGGRAPH 2011)[1]
- Foley's PhD thesis about Spark [2]
- Spark user's guide: [3]

The initial code base is available in[4]. The research paper about Spark summarizes high-level considerations and core system details, which may also be found more detailed in Foley's PhD thesis. The user's guide gives an introduction to the Spark shading language. The main part of this thesis may be considered as a case study of Spark, as a *source-to-source* compiler. Within this thesis, the (above mentioned) existing documentation is extended with supplementary low-level discussion about each phase of the compilation process. Usually, the implementation of a compiler does not require any mathematical computations and simply involves the creation and processing of intermediate representations (IR). The data structures, their initialization and the processing of these IR are central. All information provided in the Chapters 5 to 8 rely on reverse engineering and the available documentation.

1.8 Structure of the Work

Given the initial thoughts, issues and goals, related work is discussed in Chapter 2.

Chapter 3 gives an introduction to the Spark shading language by examining some examples and language features.

Chapter 4 provides some informal and practical basics of language processing.

Chapter 5 provides some details about the data structures and syntax analysis in Spark.

Chapter 6 presents data structures and algorithms of the semantic analysis phase. Some stylistic peculiarities and a common implementation theme is highlighted.

[1] `http://graphics.stanford.edu/papers/spark/spark_preprint.pdf`

[2] `http://graphics.stanford.edu/~tfoley/papers/tfoley-dissertation.pdf`

[3] `http://cloud.github.com/downloads/spark-shading-language/spark/spark-users-guide.pdf`

[4] `https://github.com/spark-shading-language/spark`

In Chapter 7 some aspects of code optimization are discussed and Chapter 8 and 9 provide details about the final HLSL and GLSL code generation phase.

Chapter 10 wraps up with an evaluation of the new shading examples and retrospective discussion on Spark.

This thesis is an extension to Foley's PhD thesis, therefore, it is highly recommended to read [3], [14] and [2] first. Chapters 1 to 3 re-iterate some ideas as needed, however, do not cover each aspect of Foley's works. While skimming over Chapters 5 to 9 the reader is advised to browse through the code to facilitate the understanding, since the description provides only a (coarse) thread through the implementation and does not cover each and every detail. Readers interested only in high-level concepts should read Chapters 1 to 3 and 10, whereas readers who are familiar with Foley's work may directly read 4 to 9. Chapter 4 may be considered as an optional crash course on language processing.

1.9 Contributions

The main contribution of this thesis is a complementary low-level documentation of the Spark framework, which is both missing in the available paper [14] and PhD thesis [3] on Spark. Furthermore, the initial Spark compiler targets the Direct3D back-end. Here, the adaption of an OpenGL 4.2 back-end is discussed and documented.

Documentation

The existing documentation (c.f. Section 1.7) is supplemented by the following low-level details. For each phase of the compiler a corresponding chapter describes certain aspects of the implementation.

- Syntax Analysis (Chapter 5):
  - Provides a closer look on the application of GPLEX/GPPG
  - Discusses key data structures of the first intermediate representation
- Semantic Analysis (Chapter 6):
  - Discusses the build system and lazy initialization mechanism used to construct the next intermediate representation
  - Essential interfaces and data structures
  - Key semantic analysis tasks: shader inheritance checks, base linearization, member declarations, member overriding, resolve statements, terms and expressions, hints on type-checking, ...
- Optimization (Chapter 7):
  - Discusses the build system and some aspects of the new intermediate representation
  - Details on how expressions are decomposed

  - Some aspects of simplification and marking output variables

- Code Generation (Chapter 8):
  - Provides details on how the final intermediate representation is used to emit HLSL code
  - Differentiates C++ (wrapper classes) and HLSL (actual shader) code generation
  - Implementation details on each pipeline stage

- OpenGL 4.2 Support (Chapter 9):
  - Discusses the most crucial changes which were necessary to adapt the different syntax and semantic of GLSL shaders
  - Differences in the wrapper classes
  - Changes to the standard library
  - Considerations on architectural issues

Codebase

Certain adaptions were made to the code generation phase to support the GLSL syntax and semantic (c.f. Chapter 9). Further, new examples were added using both the Direct3D and OpenGL back-end:

- HLSL examples
  - SimpleHLSL: simple shading example showing the application of Spark without the noise of additional libraries (DXUT, etc)
  - SimpleBezier11: adaptive tessellation of a bezier surface

- GLSL examples
  - SimpleGLSL: simple shading example using the OpenGL back-end
  - CubeMapGLSL: dynamic cube mapping
  - SimpleBezierGLSL: adaptive tessellation of a bezier surface

In Section 10.1, the new OpenGL back-end is evaluated by comparing Spark shaders to custom GLSL shaders. To measure the exact run-time deviations the application-side differences are reduced to a minimum.

CHAPTER 2

Research Analysis

The amount of scientific work addressing shader development is substantial. Ideas of shading languages and programmable shading existed even before the introduction of accelerated graphics hardware. Due to rapid development in hardware architectures, shading languages and APIs, current challenges differ widely from difficulties and limitations in the past. Here, the most practical and up-to-date approaches will be discussed and compared to the Spark framework.

2.1 Related Work

Effect Frameworks

Effect frameworks facilitate the encapsulation of the graphics state and shader code. In one of the first articles about effect frameworks, the author motivates the usage of effect frameworks and so-called *effect files* as an alternative to the usual *individual file* approach, where each shader is placed in a separate file [12]. Microsoft's effect framework (2003) and NVIDIA's CgFX (2003) were first implementations. The Direct3D 11 effect framework[1] allows the description of *groups*, *techniques* and *passes*. Groups are optional and define a set of techniques. For example, a collection of material shaders could be grouped in an effect file. A technique, however, defines a set of rendering passes and each pass defines necessary pipeline states to render an effect. Figure 2.1 shows relevant code fragments of a basic Direct3D 11 effect file, which contains (conceptual) global variables, state specifications, shader code and technique descriptions. Two single-pass techniques are specified:

- RenderSceneWithTexture1Light: comprises a vertex and pixel shader
- RunComputeShader: uses a compute shader

[1] `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476136(v=vs.85).aspx`

```
// Global variables
// ...

// Texture and sampler state
// ...

// Other non-programmable pipeline state
// ...

// Shaders
//   RenderSceneVS, RenderScenePS and CS
// ...

// Groups, Techniques and Passes
technique10 RenderSceneWithTexture1Light
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, RenderSceneVS( 1, true, true ) ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, RenderScenePS( true ) ) );
    }
}

fxgroup g0
{
    technique11 RunComputeShader
    {
        pass P0
        {
            SetComputeShader( CompileShader( cs_5_0, CS() ) );
        }
    }
}
```

Figure 2.1: HLSL effect framework

The compute shader technique is placed in a *fxgroup*, anyhow, the RenderSceneWithTexture1Light is a global technique and is implicitly placed in an unnamed group.

A more elaborate example could contain several techniques with multiple passes. For example, a material shader could be specified using several techniques, each corresponding to a fallback shader for certain hardware capabilities. Different passes could be used to render usual geometry in the first pass and to add sprites in a follow-up pass.

Usually, following pipeline states have to be provided:

- Shader state: vertex-, hull, domain, geometry, pixel (and compute) shader
- Texture and sampler state
- Other non-programmable pipeline state

Non-programmable pipeline stages are configured using structures:

- D3D11_RASTERIZER_DESC: defines the state of the rasterizer
- D3D11_BLEND_DESC and D3D11_DEPTH_STENCIL_DESC: define the state of the output merger and depth buffer

```
// File SimpleBezier11.cpp
// ...
D3D_SHADER_MACRO integerPartitioning[] = { { "BEZIER_HS_PARTITION", "\"integer\"" }, { 0 } };
D3D_SHADER_MACRO fracEvenPartitioning[] = { { "BEZIER_HS_PARTITION", "\"fractional_even\"" }, { 0 } };
D3D_SHADER_MACRO fracOddPartitioning[] = { { "BEZIER_HS_PARTITION", "\"fractional_odd\"" }, { 0 } };

V_RETURN( CompileShaderFromFile( L"SimpleBezier11.hlsl", NULL, "BezierVS",
                                 "vs_5_0",  &pBlobVS ) );
V_RETURN( CompileShaderFromFile( L"SimpleBezier11.hlsl", integerPartitioning, "BezierHS",
                                 "hs_5_0", &pBlobHSInt ) );
V_RETURN( CompileShaderFromFile( L"SimpleBezier11.hlsl", fracEvenPartitioning, "BezierHS",
                                 "hs_5_0", &pBlobHSFracEven ) );
V_RETURN( CompileShaderFromFile( L"SimpleBezier11.hlsl", fracOddPartitioning, "BezierHS",
                                 "hs_5_0", &pBlobHSFracOdd ) );
V_RETURN( CompileShaderFromFile( L"SimpleBezier11.hlsl", NULL, "BezierDS", "ds_5_0", &pBlobDS ) );
V_RETURN( CompileShaderFromFile( L"SimpleBezier11.hlsl", NULL, "BezierPS", "ps_5_0", &pBlobPS ) );
V_RETURN( CompileShaderFromFile( L"SimpleBezier11.hlsl", NULL, "SolidColorPS", "ps_5_0", &pBlobPSSolid ) );
// ...
```

```
// File: SimpleBezier11.hlsl
//...
#ifndef BEZIER_HS_PARTITION
#define BEZIER_HS_PARTITION "integer"
#endif // BEZIER_HS_PARTITION
//...
[partitioning(BEZIER_HS_PARTITION)]
//...
```

Figure 2.2: Code Fragments of the SimpleBezier11 example from the DirectX SDK (June 2010)

Shader state changes require a more precise control, therefore a finer granulation into

- Constant buffer state
- Sampler state
- Shader resource state
- And unordered access view state (for pixel and compute shaders)

is provided.

Effect files enhance the manageability of shaders and allow the implementation of fallback shaders. The pipeline state is specified more cleanly. However, shader combination or shader inheritance is not supported.

Über-Shader

The *über-shader* implements several predefined effects in a single file. Certain features may be modified using the *preprocessor*. Figure 2.2 shows code fragments of a basic über-shader implementation, which can be found in the examples of the DirectX Software Development Kit (SDK)[2]. The application implements a simple tessellation algorithm of a bézier surface. In the

[2]DirectX SDK (June 2010): `http://www.microsoft.com/en-us/download/details.aspx?id=6812`

```
class Diffuse : public ShaderBase<Diffuse, IlluminatedMaterial>
{
public:
     void_<> illuminance(vec3<>, vec3<>)
     { return invoke< void_<> >("Diffuse_illum_impl"); }

     ValueReference<vec4, uniform> color;

private:
     DERIVED_DECL(Diffuse, IlluminatedMaterial)
};

CLASS_INIT(Diffuse, "Diffuse.glsl", NONE, DEFS((color)) )
```

```
// Diffuse.glsl
void Diffuse_illum_impl(Diffuse_SELF, vec3 light_color, vec3 light_direction)
{
     vec3 normal = IlluminatedMaterial_get_normal(self);
     float intensity = max(0., dot(light_direction, normal));
     vec4 color = intensity * vec4(light_color, 1) * Diffuse_get_color(self);
     IlluminatedMaterial_accum_color(self, color);
}
```

Figure 2.3: Coupling of proxy objects and GLSL shading code

hull shader different subdivision modes (integer, fractional even, fractional odd) may be specified. In the application three different versions of the hull shader are compiled, each corresponding to a different subdivision mode. Later, the user may switch between different tessellation modes. The über-shader allows simple variation and combination of shaders, without producing duplicate shader code. Also, features may be enabled or disabled using the preprocessor and data-dependent control flow. However, programmers are forced to work on a single complex shader.

Object-Oriented Shader Design

In [3, Section 2.1.5] Foley refers to effect systems and über-shaders as *shader metaprogramming* concepts. Kuck and Wesche present an innovative and recent metaprogramming approach, which addresses the shader combination problem [9] [13]. They introduce a lightweight object-oriented framework on top of the OpenGL 3.2 pipeline and GLSL for the standard illumination situation (surfaces illuminated by light sources), consisting of two dependent parts:

- An object system for GLSL and
- proxy objects in C++

Classes are declared and instantiated in the application. Class attributes and methods correlate to pipeline stages by exposing stage qualifiers. For each instanced object, its attributes are declared as global variables in GLSL. Figure 2.3 shows how shading code is coupled to proxy objects. A `Diffuse` class is implemented by deriving from a `IlluminatedMaterial` base class, which is used by materials that receive light from light sources. Without going into details,

note that a uniform color attribute is declared and an `illuminance` method is implemented. The illuminance method itself only holds the name of the actual GLSL method in *Diffuse.glsl*. The GLSL implementation uses methods of the base class to retrieve the normal vector and to accumulate the material color. Kuck and Wesche propose a novel reference type to map application classes to GLSL, which allows

- the reference of objects as variables and
- calling methods of objects in the shading language.

In GLSL objects are mapped to plain numbers. Dispatch functions are created for all attributes and methods of the instanced classes. The complete GLSL shader comprises the GLSL implementations and the dispatcher functions. Classes are created in the application stage, which facilitates the allocation of GPU resources. A single root object provides the entry point for all stages and triggers necessary method calls of referenced objects.

Figure 2.4[3] shows a typical calling sequence in Kuck and Wesche's framework. The root `Surface` object, which is not shown in the sequence diagram, holds references to `Material`, `Light` and `CoordinateSystem` objects. In the vertex shader the coordinate system is initialized, where necessary data is transformed into the coordinate system where lighting is performed. Then the material is initialized, which triggers the transformation of the light sources. First the `transform_light` method is called on the `CoordinateSystem` object by providing the list of light sources. The `CoordinateSystem` object then calls corresponding `transform_light` methods on the actual light sources, by providing a self reference. In the fragment shader the `shade` method is invoked, which triggers `illuminate` calls on each light source. The shading is calculated indirectly by a double dispatch procedure, which allows various shading effects for different light and material types.

2.2 Comparison

Effect Frameworks

Effect frameworks, in comparison to plain GLSL or HLSL shaders, provide a more consistent encapsulation of the hardware state setup and shading code. Similar shading techniques may be grouped into *fxgroups* and several fall-back shaders may be provided by supporting numerous techniques of a single effect. High-end computers may benefit from more appealing visual effects by evaluating expensive techniques, whereas older hardware generations may still run lower-cost implementations. Multiple passes may be defined in a technique, which enhances the manageability of the shading code and state changes. The Direct3D 11 effect framework simply extends the HLSL syntax to support groups, techniques and passes. Therefore, learning and using effect files is straightforward. However, problems like shader combination, extension, modularity and code reusability still persist.

[3]The sequence diagram was adapted from Kuck's paper

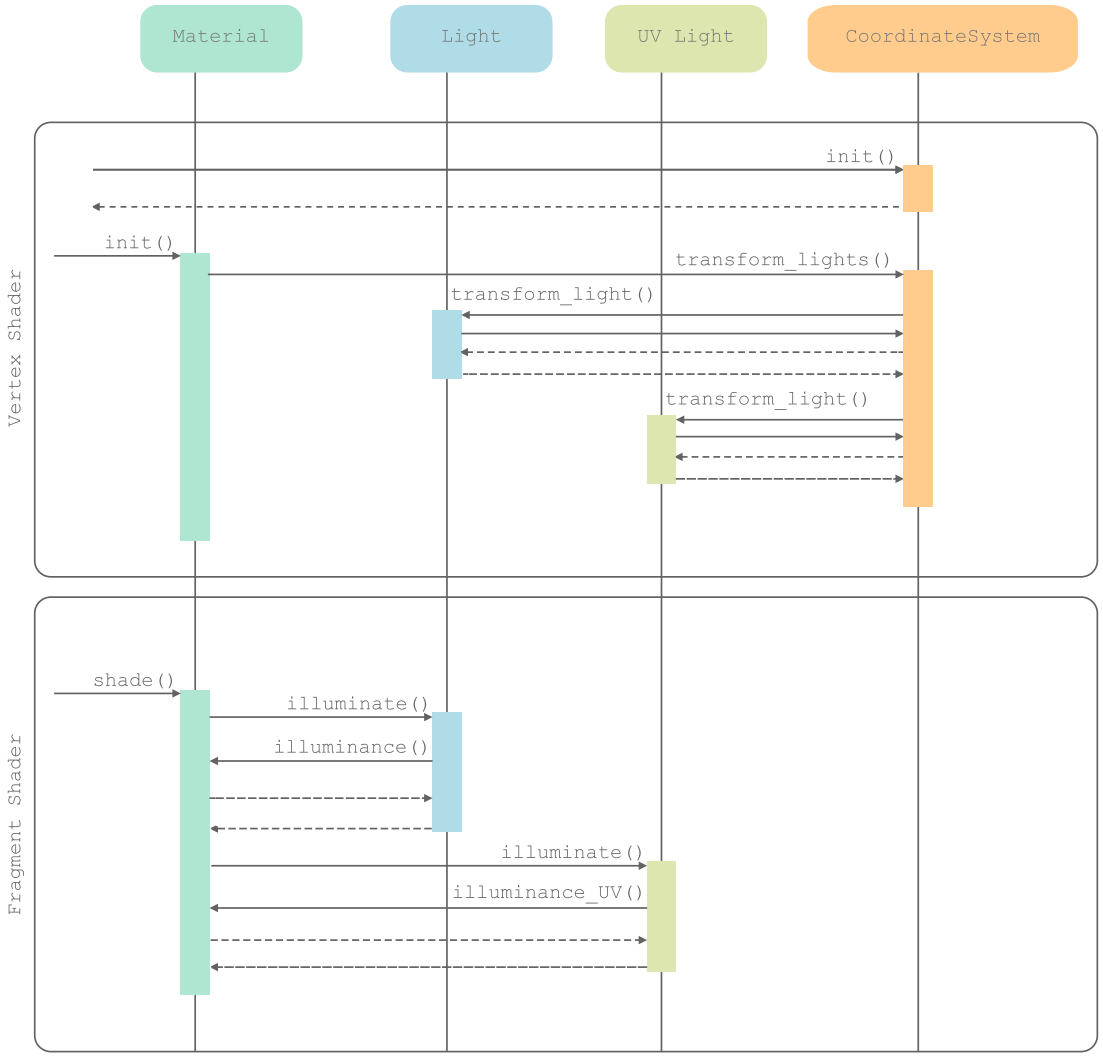

Figure 2.4: Shader metaprogramming sequence diagram

Über-Shader

The über-shader implements all desired effects in a single and probably complex file. With the help of the preprocessor and data-dependent control flow, predefined features may be enabled or disabled. Also simple shader variations may be achieved. The HLSL or GLSL shader code with the according preprocessor evaluation is compiled. At run-time, the user may switch between the provided shader programs. Similar to effect frameworks, also fall-back shaders may be implemented by integrating different preprocessor definitions for different hardware targets. Multipass rendering and state encapsulation is not considered. Since the über-shader comprises the implementation of every preprocessor definition in a single file, teamwork and code maintainability is aggravated. Also, the actual development issues, similarly to effect files, are still inherent.

Object-Oriented Shader Design

Kuck and Wesche present another metaprogramming framework for the OpenGL 3.2 pipeline, which improves shader combination, extension and code reusability. The shading effect is defined by incorporating C++ code and the GLSL object system. From the developer point of view, the actual shading code is still specified in GLSL, since the C++ method definition merely holds the name of the GLSL procedure. The final GLSL shader comprises shading code, global variables for all object attributes and necessary dispatcher methods. Several effects may be implemented for the standard illumination situation with enhanced code reusability and shader combination possibilities. However, the framework brings additional compilation- and predictable runtime costs. Currently, there is no follow-up work or adaption for the actual rendering pipeline, including tessellation and compute shaders.

2.3 Spark Background

In [3, Chapter 2] Foley outlines the evolution of real-time shading languages and mentions approaches, which influenced his design decisions. Although shader programming is practically used since 2001, the ideas of a shading language go back to the mid 1980s. Back then two opposite approaches emerged.

- Cook's shade trees
- Perlin's image synthesizer

Cook proposed a *declarative language*, where surface, light, atmosphere and displacement shaders are constructed as separate graphs. While these graphs impose highly structural properties, which make shader combination possible, certain constraints have to be followed. Mutable variables or control flow are not available. On the other side Perlin suggested a *procedural approach*, where almost any algorithm may be expressed, while exposing only a procedural structure. Further, Foley states that modern shading languages often derive either from Cook's declarative language or Perlin's procedural approach. He mentions the RenderMan Shading

Language (RSL) as a mixture of both possibilities. In particular, the idea of a shader as an *instantiated object* of an object-oriented class influenced the design of Spark. Thereby, expensive operations on shaders (e.g. specialization) and shader lifetime can be managed properly from the application point of view. Although RSL being an procedural language, surface and light shaders are separated. Interaction between shaders is facilitated by an interface via illuminance loops and illuminate calls. Furthermore, RSL introduces the concept of *computational rates*. Values are specified per-batch or per-sample, e.g. a directional light vector as a *uniform variable* and diffuse reflectance of the surface with *varying rate*. The Stanford Real-Time Shading Language (RTSL) enhances the set of rate qualifiers. A single *per-pipeline shader* may target both vertex and fragment stages by partitioning computations at *vertex* or *fragment rate*. Spark adapts computation rates as an *extensible concept* and applies these ideas to the modern graphics pipeline.

CHAPTER 3

Introduction to Spark

Spark is a *per-pipeline* shading language which merges various aspects of other languages and software engineering notions to improve real-time shader programming. Foley provides a user's guide [2] to quickly start developing Spark shaders. As mentioned, in his PhD thesis he provides an overview of many high-level concepts [2].

However, this chapter presents some facets of Spark by providing examples using the novel OpenGL 4.2 back-end. The entire code base with the OpenGL 4.2 back-end and additional examples are available on github[1].

3.1 Basic Example

Figure 3.1 shows the simple diffuse lighting and texturing effect from Chapter 1 (Figure 1.3) implemented in a single shader class.

Shader Classes

The shading code is comprised in a shader class. Forget about the implementation details for a second. Basically, all declarations and definitions have to be located in shader classes. Basic concepts known to developers familiar with object-oriented programming also apply to Spark shaders. A shader class may derive from other classes and inherit the definitions and declarations of the base class. Abstract declarations and virtual definitions may be used to provide interfaces and allow customizability.

The `BasicSpark` class *inherits* types, methods and other features from the OpenGL 4.2 pipeline by *deriving* from the `OpenGL42DrawPass` base class, which is located in the *standard library* also written in Spark.

Programmers may opt between OpenGL 4.2 and Direct3D 11 by either deriving from `OpenGL42DrawPass` or `D3D11DrawPass`.

[1] `https://github.com/dinony/spark_opengl`

```
shader class BasicSpark extends OpenGL42DrawPass
{
    // @Uniform
    input @Uniform mat4 world;
    input @Uniform mat4 view;
    input @Uniform mat4 proj;
    input @Uniform vec4 w_lightPos;
    input @Uniform sampler2D tex;

    @Uniform mat4 modelView = view * world;
    @Uniform mat4 mvp = proj * modelView;
    @Uniform mat3 normalMatrix = mat3(transpose(inverse(modelView)));

    // @AssembledVertex
    input @AssembledVertex vec3 m_pos;
    input @AssembledVertex vec3 m_normal;
    input @AssembledVertex vec2 texCoord;

    // @CoarseVertex
    @CoarseVertex vec4 v_pos = modelView * vec4(m_pos, 1.0f);
    @CoarseVertex vec3 v_pos3 = v_pos.xyz / v_pos.w;

    @CoarseVertex vec4 v_lightPos = view * w_lightPos;
    @CoarseVertex vec3 v_lightPos3 = v_lightPos.xyz / v_lightPos.w;

    @CoarseVertex vec3 v_lightDir = normalize(v_lightPos3 - v_pos3);
    @CoarseVertex vec3 v_normal = normalMatrix * m_normal;

    // @RasterVertex
    override RS_Position = mvp * vec4(m_pos, 1.0f);

    // @Fragment
    @Fragment vec4 diffuse = texture(tex, texCoord);
    @Fragment float lighting = max(0.0f,
         dot(normalize(v_lightDir), normalize(v_normal)));
    @Fragment vec4 color = diffuse * lighting;

    // @Pixel
    output @Pixel vec4 myTarget = color;
}
```

Figure 3.1: Basic Spark Shader (Colors denote different computation rates)

Attributes

Another characteristic of Spark are the attribute definitions:

```
input @Uniform mat4 world;
```

Programmers familiar with GLSL will assume a uniform variable declaration. Basically, a more precise *frequency-qualified* type declaration is provided. `@Uniform` defines the computation rate and `mat4` defines the actual type the world variable will acquire at run-time.

But what about the following uniform variable definition?

```
@Uniform mat4 modelView = view * model;
```

A uniform variable definition, which is initialized by other uniform variables. Actually, the Spark compiler recognizes such variables and generates C++ code, which will initialize the according `modelView` variable before it is sent to the GPU.

```
@Fragment vec4 diffuse = texture(tex, texCoord);
```

The above attribute definition can be read as `diffuse` being a variable of type `vec4`, which is evaluated *per-fragment*. That is, the compilation process has to map `diffuse` to the fragment shader. Further, the per-assembled-vertex `texCoord` variable has to be plumbed through all intermediate stages. That is, a type conversion routine is required to transform `@AssembledVertex T` to `@Fragment T` types (T being a generic type parameter). In Spark, pre- and user-defined *plumbing operators* are used to configure the compiler to perform the required conversions.

Per-pipeline Shader

In contrast to current per-stage languages, several programmable stages may be targeted in a single per-pipeline shader class. There is much more to mention about Spark, however, further details are provided when new examples are introduced.

3.2 C++ Bindings

Spark shaders are translated to GLSL or HLSL shaders. Additionally, the compiler generates wrapper classes to interface with the C++ application.

Figure 3.2 shows the basic structure of a C++ application using Spark. At the beginning a Spark context is acquired. Then the usual tasks are performed:

- Initialize GLFW
- Create a window and an OpenGL context
- Initialize GLEW

```
#include <iostream>
// Other includes...

#include "GL/glew.h"
#include "GL/glfw.h"
// Also include glm...

// SPARK:
#include "BasicSpark.spark.h"

// SPARK:
spark::IContext* gSparkContext = nullptr;
BasicSpark* gShaderInstance = nullptr;
// Other global variables...

int main()
{
  // SPARK:
  gSparkContext = SparkCreateContext();

  // glfw initialization...
  // Create a window...
  // glew initialization...
  // Geometry initialization...
  glm::mat4 proj = glm::perspective(75.0f, aspect,
                       1.0f, 1000.0f);
  // Other matrices etc.

  // SPARK:
  gShaderInstance =
     gSparkContext->CreateShaderInstance<BasicSpark>(0);
  GLint vertexLoc = gShaderInstance->getPosLocation();
  // Get other locations ...

  // Create vertex buffers
  // Create vertex array objects
  // Buffer geometry on gfx ram
  // Load textures
  // Setup graphics state

  while (running)
  {
     // Do input handling, animation, etc...

     // SPARK:
     // Update per-frame shader uniforms
     gShaderInstance->SetProj(proj);
     // ..

     // Setup DrawSpan

     glClear( .. | .. );
     gShaderInstance->Submit();
     glfwSwapBuffers();
  }
  // SPARK:
  if(gShaderInstance) gShaderInstance->Release();

  // Clean up rest
  glfwTerminate();
  return 0;
}
```

Figure 3.2: Basic Spark Application

- Initialize matrices, camera, projection, etc

With the Spark context the shader is created. As usual, the vertex buffer objects and vertex array objects are initialized. In the rendering loop shader inputs are updated via automatically generated setter methods. Finally, the shader is executed by calling `Submit(..)`. From the application point of view the most basic Spark example is shorter than the analog OpenGL example. Behind the scenes the following tasks are adopted by Spark :

- During initialization:
  - Create and compile the generated GLSL shaders
  - Shader error recovery
  - Create and initialize the according uniform buffer object
- In the render loop:
  - Perform constant and uniform computations
  - Update uniform buffer

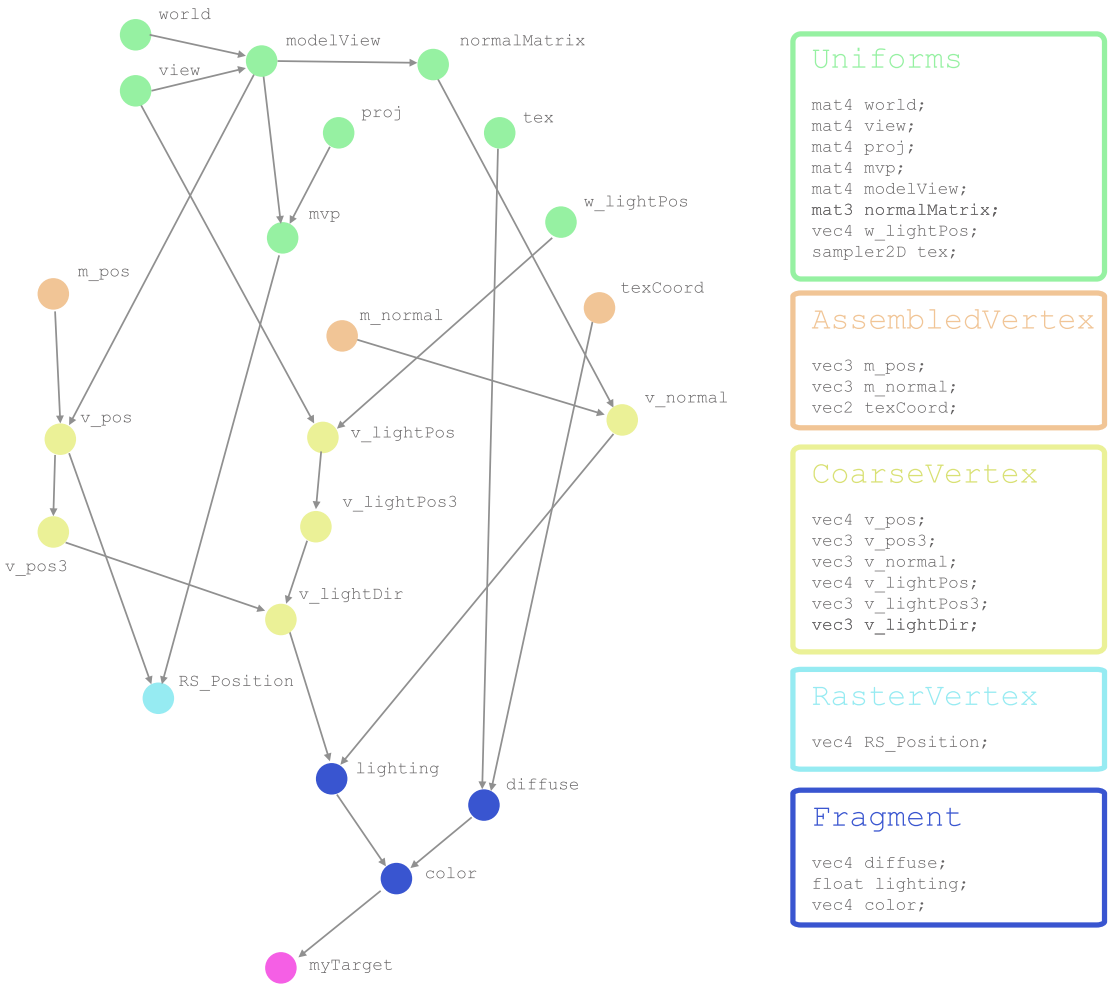

Figure 3.3: Shader Graph (left), Record Types (right) of the BasicSpark example

- Bind VAO and call according draw method

3.3 Spark Design Goals

A basic Spark example has been presented. Here, some thoughts and background knowledge are provided, which influenced the language design decisions and the implementation of the compiler.

Declarative and Procedural Aspects

Spark unifies declarative and procedural features [3, Section 3.3.1]. While a class definition exhibits declarative properties, in procedural subroutines modern programming constructs like

control flow and loops are available.

Each attribute in the class declaration represents a node in the *shader graph*, which is shown on the left side of Figure 3.3. Spark code is translated to procedural HLSL or GLSL shaders, so the computations have to be mapped to according pipeline stages. This is achieved via the *1-to-1 relationship* between computation rates and *record types* in the generated per-stage procedures.

Each graph-node corresponds to a certain computation rate and defines a field in the according record type. Shader stages communicate via record types, for instance, the vertex shader receives a stream of `AssembledVertex` records to apply per-(coarse)vertex computations.

Edges connecting nodes with different colors denote changes in the computation rate. therefore, *plumbing operators* have to be defined. For instance, when a per-vertex color attribute is used in a per-fragment computation, the values are interpolated.

The pipeline implicitly implements some plumbing operators, for instance, the interpolation from raster vertices to fragments by the fixed-function rasterizer. Custom plumbing operators may be defined to provide explicit conversion operators for certain types.

The shader graph defines only *point-wise* shading code, while the current graphics pipeline supports procedural group-wise operations (c.f. Section 1.4). In Spark, group-wise operations are specified in shader subroutines. These methods may be used to define attributes. Also, plumbing operators may involve group-wise computations, for instance, when interpolating fine vertex attributes for given parametric locations.

Plumbing operators are implemented as special subroutines with explicit rate qualifiers and without control flow. In particular, plumbing may introduce intermediate results and additional attributes (graph-nodes).

Shaders as Classes

The modeling of shaders as classes is beneficial [3, Section 3.3.2]. Mechanisms for modularity and composition are exposed via well-known object-oriented principles. Using virtual nodes, customizable parts are defined, whereas abstract classes may declare interfaces. Shader classes may derive from other classes to inherit attributes and specialize by adding new graph-nodes. Composition of two and more shaders is supported by *mixin inheritance* concepts known from other languages. To avoid problems associated with multiple inheritance, the C3 linearization approach is implemented to flatten the inheritance tree. However, it is crucial to determine a total order of the inherited bases to resolve virtual members. Most linearization concepts bring some drawbacks [3, Section 2.3.1]. Beside name-clashing issues for dynamic languages, changing the inheritance order might change the behavior of the program, which is, generally, problematic for modularity.

From the application point of view, shader creation, management and execution can be handled much better. There is a clean *phase separation* of heavy-weight operations like compiling and light-weight shader execution.

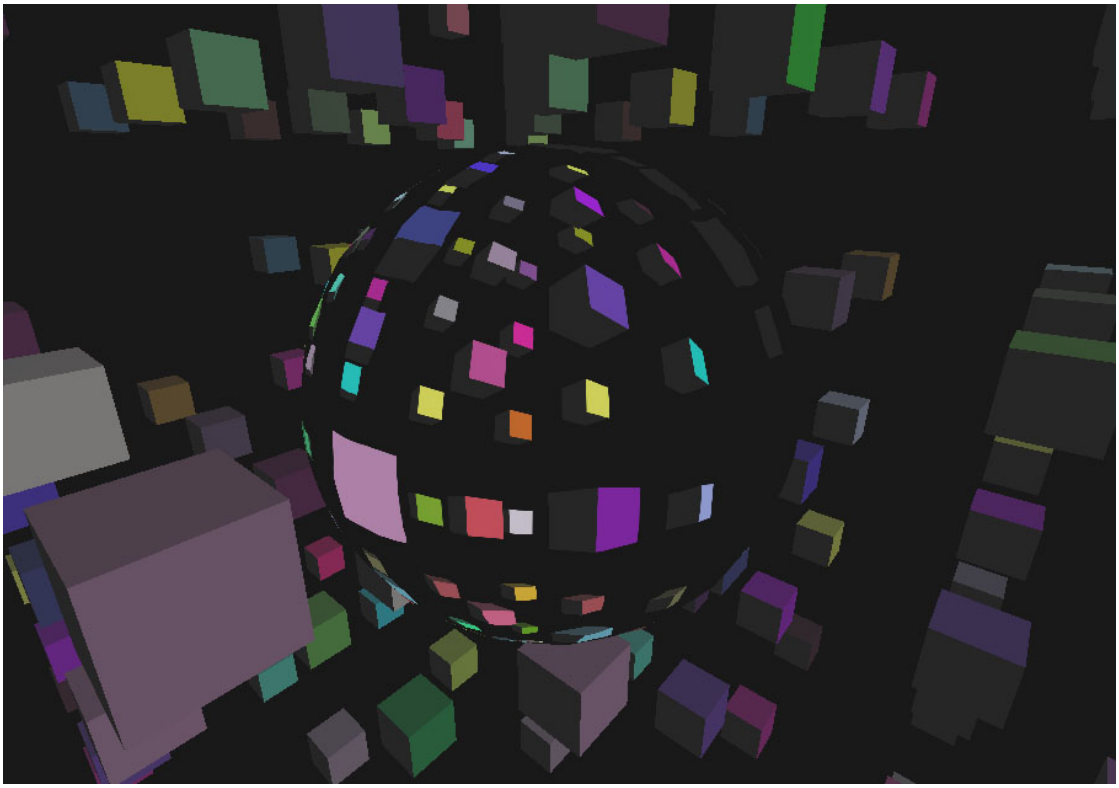

Figure 3.4: Dynamic Cube Mapping in Spark (OpenGL 4.2 back-end)

3.4 Dynamic Cube Mapping

Cube mapping is a popular algorithm to render reflective surfaces in real-time. When the environment changes or the reflective object is moving the cube maps have to be updated dynamically. Figure 3.4 shows a scene consisting of a grid of randomly colored cubes and a moving reflective sphere. There are several possible implementations:

- CPU Multipass implementation:
  - Generate a cube map texture
  - Six passes are used to create the cube map
  - Each pass sets the view direction for the current face of the cube map
  - Seventh pass renders the scene from the viewer's perspective
  - An additional pass to perform cube mapping
- Geometry shader:
  - Use the geometry shader to render to the different faces of the cube map

  - Six passes are relocated to the GPU
  - Only three CPU passes needed

- Geometry shader instancing:
  - Invoke geometry shader six times
  - Render to the different faces of the cube map simultaneously

Here, the implementations using the geometry shader are discussed. The abstract class `Base` in Subsection 1.6 (c.f. Figure 1.3) implements the transformation of a vertex. Also, the `Diffuse` class implements a simple lighting model. Therefore, these two classes may be reused to render the scene. Now, the real interesting part is the shader to create the cube map.

The following abstract class may serve as the base class for different render to cube map shaders (only crucial parts are shown):

```
abstract shader class RenderToCubeMapBase
    extends OpenGL42DrawPass, OpenGL42GeometryShader
{
    // ...
    input @Uniform Array[mat4, 6] views;

    // We are always taking one triangle as input
    override GS_InputVertexCount = 3;
    input @RasterVertex int vertexID;
    input @RasterVertex int faceID;
    @RasterVertex FineVertex fVertex = GS_InputVertices(vertexID);
    @RasterVertex mat4 curView = views(faceID);
    @RasterVertex mat4 curViewProj = proj * curView;
    @RasterVertex mat4 curMvp = curViewProj * world;
    @RasterVertex mat4 curModelView = curView * world;
    @RasterVertex mat3 normalMatrix =
         mat3(transpose(inverse(curModelView)));

    @RasterVertex vec4 m_pos = vec4(m_vertex, 1.0f);

    override RS_Position = curMvp * m_pos;

    override @RasterVertex T FineToRaster[type T](@FineVertex T value){
        return value @ fVertex;
    }
}
```

`RenderToCubeMapBase` derives both from `OpenGL42DrawPass` and

`OpenGL42GeometryShader`. Every render to cube map shader requires an array of view matrices corresponding to the view directions. The geometry shader always works on three vertices (triangles) and projects each vertex to the according cube map face. Notice the two `@RasterVertex vertexID` and `faceID`. To perform the per-raster-vertex operations these two attributes have to be provided. That is, the current cube map face has to be known and the vertex number of the current triangle. The vertex id is used to fetch the current vertex from the array of input vertices:

```
@RasterVertex FineVertex fVertex = GS_InputVertices(vertexID);
```

and the face id is used to fetch the current view matrix:

```
@RasterVertex mat4 curView = views(faceID);
```

Notice the custom plumbing operator:

```
override @RasterVertex T FineToRaster[type T]( @FineVertex T value ){
    return value @ fVertex;
}
```

Remember that the operator `()` is used for array access and `[]` is used to specify generic parameters in Spark [2, Section 2.7].

`@FineVertex` attributes max be projected out of the `fVertex` attribute [2, Section 2.6].

An actual implementation derives fom `RenderToCubeMapBase` and overrides and implements all abstract or virtual attributes and method declarations, respectively.

```
shader class RenderToCubeMap
    extends RenderToCubeMapBase, OpenGL42NullTessellation
{
    override GS_InstanceCount = 1;
    override GS_MaxOutputVertexCount = 18;

    override @GeometryOutput void GeometryShader()
    {   // for all views
        for(j in Range(0,6))
        {   // render all vertices
            for(i in Range(0,3)){
                EmitVertex(RasterVertex(vertexID : i, faceID : j));
            }
            EndPrimitive();
        }
    }
}
```

`GS_InstanceCount` denotes that there is only a single invocations of the geometry shader. `GS_MaxOutputVertexCount` specifies that 18 vertices are produced by the `GeometryShader()` method. Anyhow, the `GeometryShader()` method, very similar to an GLSL geometry shader, iterates over all view matrices and over each triangle vertex and performs the `@RasterVertex` computations, which are common to all render to cube map variations (in `RenderToCubeMapBase`).

Another implementation uses geometry shader instancing:

```
shader class RenderToCubeMapGSInst
    extends RenderToCubeMapBase, OpenGL42NullTessellation
{
    override GS_InstanceCount = 6;
    override GS_MaxOutputVertexCount = 3;

    override @GeometryOutput void GeometryShader()
    {
        for(i in Range(0,3)){
            EmitVertex(RasterVertex(vertexID : i
                       , faceID : gl_InvocationID));
        }
        EndPrimitive();
    }
}
```

The `GeometryShader()` method is invoked for each view matrix. The geometry shader, however, produces only three vertices per invocation. Again notice the `RasterVertex` record type constructor call.

```
RasterVertex(vertexID : i, faceID : gl_InvocationID)
```

Calling the record type constructor has the effect of performing all calculation for the given computation rate [3, Section 4.1.3].

3.5 Distance Adaptive Tessellation

Distance adaptive tessellation is implemented in Spark to present the support for tessellation. In the previous section the geometry shader was accessed by deriving from `OpenGL42Geometry` shader. When the geometry shader is active and tessellation stages are skipped, the standard library provides the `OpenGL42NullTessellation` class to provide a default plumbing operator used to convert per-coarse vertex attributes to per-fine vertex values.

Distance adaptive tessellation is a simple algorithm to adjust the level of detail needed for rendering appealing images. Basically, in the tessellation control shader the distance of the current triangle or quad patch is computed to determine the inner and outer tessellation levels. Of course, there are other criteria to determine the level of tessellation, however, here a simple example is presented to show how tessellation shader are implemented in Spark.

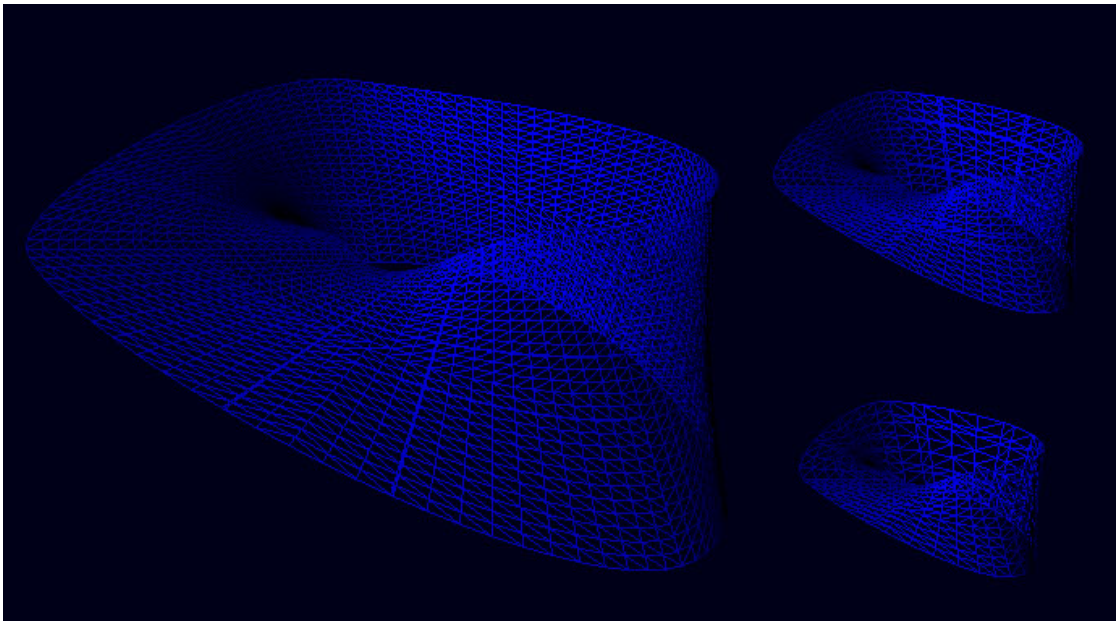

Figure 3.5: Distance Adaptive Tessellation of a Bezier Surface (OpenGL 4.2 back-end)

Figure 3.5 shows a tessellated bezier surface. The initial approximation of the möbius strip is given by four quads. From the GLSL point of view the tessellation control shader sets the tessellation levels, and the tessellation evaluation shader comprises the cubic Bezier surface evaluation code for the generated parametric locations.

The Spark implementation must declare a shader class deriving from `OpenGL42DrawPass` and `OpenGL42QuadTessellation`:

```
shader class AdaptCubicBezierTess
    extends OpenGL42DrawPass, OpenGL42QuadTessellation
{ // ... }
```

`OpenGL42QuadTessellation` derives from the standard library shader class `OpenGL42Tessellation` and provides a specialization for quad tessellation (`OpenGL42TriTessellation` is available for triangle tessellation). The base class declares a couple of abstract constants used by the tessellation stages:

```
abstract output @Constant int HS_OutputControlPointCount;
abstract output @Constant TessellationDomain TS_Domain;
abstract output @Constant TessellationPartitioning TS_Partitioning;
abstract output @Constant TessellationOutputTopology TS_OutputTopology;
// etc ...
```

The `AdaptCubicBezierTess` implementation has to provide these constants to configure the TPG. Besides, `OpenGL42Tessellation` defines plumbing operators required to convert between the record types specified in [2, Section 3.8].

Tessellation Control Shader

Lets break up the Spark implementation to make it clearer for traditional GLSL programmers. When working with tessellation shaders usually control points are given. So, in Spark there is a corresponding rate of computation. The standard library declares:

```
// in stdlib.spark class: OpenGL42Tessellation
input @__InputPatch
   Array[CoarseVertex, HS_InputCoarseVertexCount] gl_in;
```

`gl_in` is an array of coarse vertices given per input patch. The `AdaptCubicBezierTess` defines a plumbing operator, which converts coarse vertices to control points:

```
implicit @ControlPoint T
    CoarseToControlPoint[type T]( @CoarseVertex T value ){
        return value @ gl_in( gl_InvocationID );
    }
```

The control points are the coarse vertices, which are projected out of the array of coarse vertices, which are given per input patch. The tessellation control shader will be evaluated for each control point in the output patch. `gl_InvocationID` corresponds to the current instance of the TCS shader. Once the control points are given the computation of the distance is simple:

```
@InputPatch vec3 aTmp = v_pos3 @ gl_in(0);
@InputPatch vec3 bTmp = v_pos3 @ gl_in(3);
@InputPatch vec3 cTmp = v_pos3 @ gl_in(12);
@InputPatch vec3 dTmp = v_pos3 @ gl_in(15);

@InputPatch vec3 v_center = (aTmp + bTmp + cTmp + dTmp) / 4.0f;
@InputPatch float d = length(v_center);

// for the given distance compute tessellation factor (tessFac)
override HS_EdgeFactor = tessFac;
override HS_InsideFactor = tessFac;
```

Tessellation Evaluation Shader

Given the parametric locations from the TPG, the tessellation evaluation shader computes the actual fine vertex positions. For the möbius strip the cubic Bezier surface evaluation requires the computation of the Bernstein basis functions:

```
vec4 bernsteinBasis(float t)
{
    float invT = 1.0f -t;
    return vec4( invT * invT * invT,
        3.0f * t * invT * invT,
        3.0f * t * t * invT,
        t * t * t );
}

vec4 dBernsteinBasis(float t)
{
     float invT = 1.0f - t;
     return vec4(-3.0f * invT * invT,
         3.0f * invT * invT - 6.0f * t * invT,
         6.0f * t * invT - 3.0f * t * t,
         3.0f * t * t);
}
```

The derivations are required to compute the tangent and bi-tangent vector to receive the normal vector for the approximated fine vertex position. The TES has access to the control points of the output patch:

```
input @FineVertex Array[ControlPoint, HS_OutputControlPointCount] gl_out;
```

Given the Bernstein basis functions the surface points are evaluated by projecting the control point out of the output patches.

```
@FineVertex vec3 evalBezier
 (@FineVertex vec4 basisU, @FineVertex vec4 basisV){
        @FineVertex vec3 result = vec3(0.0f, 0.0f, 0.0f);
        result = basisV.x *
           ((v_cp3 @ gl_out(0)) * basisU.x +
            (v_cp3 @ gl_out(1)) * basisU.y +
            (v_cp3 @ gl_out(2)) * basisU.z +
            (v_cp3 @ gl_out(3)) * basisU.w );

        result = result + ...; // next 4 control points...
        return result;
    }
```

CHAPTER 4

Language Processing Basics

4.1 Compilation Process

A compiler is a program that translates a given *source program* into a *target program* without losing the semantic equivalency [1, Section 1.1]. A source program describes data structures and according computations in the *source language*. The compiler, as a *language processor*, translates the source language to the *target language*. Eventually, executable machine code is produced, which processes inputs and produces appropriate outputs. Figure 4.1 shows that the complete translation process also involves other programs: the *preprocessor*, *assembler*, *linker* and the *loader* [1, Section 1.1]. Usually, the source program is spread over multiple files. The preprocessor collects all code fragments and expands macro definitions. The compiler receives the collected and modified source program and outputs an assembly program. The assembler, for the first time, produces *relocatable machine code*. More complex programs usually rely on libraries and external tools. The linker resolves issues regarding external memory addresses and allows the reusability of other relocatable programs. Finally, the loader places the complete executable program into the memory.

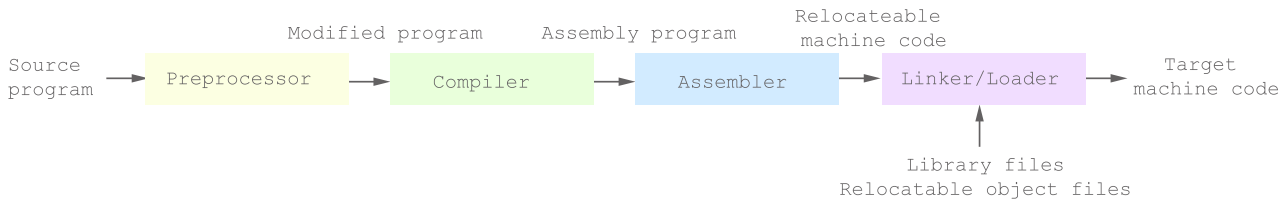

Figure 4.1: Language processing system

4.2 Compiler Structure

The compiler is divided into an *analysis* and *synthesis* part [1, Section 1.2]. The compiler receives the source program as a *character stream*. The analysis front-end processes the source program and extracts syntactic and semantic properties, whereas the synthesis back-end produces the target program.

Lexical Analysis

The compiler receives a character stream. In the first phase of analysis, meaningful character groups, *lexemes*, are extracted by *lexical analysis (scanner)*. Each lexeme belongs to a certain class of *tokens*. A token declares a class of meaningful character sequences, which are expressed using *regular expressions*.

```
tmp = x * 2;
```

The above code line could be partitioned into the lexemes:

```
"tmp", "=", "x", "*", "2" and ";"
```

Obviously, the lexemes `"tmp"` and `"x"` belong to the same class of tokens denoting identifiers. The lexeme `"2"` is recognized as another token denoting a digit or integer number. All in all, a token and a semantic value is generated for each recognized lexeme. The semantic value, of course, depends on the token type. In the case of a simple digit, the semantic value is the digit itself, whereas the semantic value of an identifier can be the character string. However, lexical analysis transfers the given character stream into a stream of tokens, which could look like:

```
<id, 1> <=> <id, 2> <*> <2><;>
```

Each token (denoted by `<..>`) comprises an *abstract symbol* and an *attribute value*. The abstract symbol declares the token type and the attribute value points to an entry in the *symbol table*, which records information about each identifier [1, Section 1.2].

Syntax Analysis

Given the token and semantic values of each lexeme, the syntax analyzer (parser) imposes a grammatical structure on the token stream. Usually, the grammar is specified by defining certain production rules in a notation similar to the *Backus-Naur form (BNF)*.

```
expr: expr '+' expr (rule 1)
expr: expr '*' expr (rule 2)
expr: TOK_NUMBER    (rule 3)
```

For example, the above production rules allow the deduction of arithmetic expressions like: $7 + 3 * 2$. Starting with the non-terminal symbol 'expr' necessary derivation steps could be

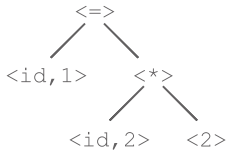

Figure 4.2: Abstract Syntax Tree (Parser)

```
expr                (start)
expr '+' expr       (rule 1)
7 '+' expr          (rule 3)
7 '+' expr '*' expr (rule 2)
7 '+' 3 '*' expr    (rule 3)
7 '+' 3 '*' 2       (rule 3)
```

The parser has to perform the reverse assignment. Given a potentially well-formed sentence, the parser has to determine necessary production rules to show that the expression can be reduced to the start symbol. That proves the conformity of the given expression and is accepted by the specified grammar. The parser constructs an *intermediate representation (IR)*, which is often referred to as the *abstract syntax tree (AST)*. Figure 4.2 shows a possible AST representation for the simple token stream from the previous section. A depth-first traversal of the AST corresponds to the evaluation of the given arithmetic expression.

Semantic Analysis

Given the AST, the semantic analyzer examines the semantic consistency of the program. Semantic analysis covers a lot of tasks, depending on the complexity of the language. Type-checking involves the validation of each operator and corresponding operands. If an operator has operands of non-matching types, a compiler error may be generated. In some cases the types can be converted. For instance, if the multiplication operator works on float values and an integer value is given, the compiler can insert additional code to convert the integer value to a floating point value.

Code Generation

After syntax and semantic analysis, usually an assembly-like IR is produced. There are several benefits for producing an assembly program [1, Section 1.1]:

- Assembler code is easy to produce and easy to translate to machine code
- Serves as an IR to optional optimizing back-ends
- Debugging is facilitated, which is crucial for developers working on a very low-level

Optionally, machine independent IR optimization may be performed, which usually leads to better machine code. There are different criteria for code optimization, for instance, faster,

```
// ...
var stream =
     new System.IO.FileStream(
          fileName,
          System.IO.FileMode.Open,
          System.IO.FileAccess.Read);

using (stream)
{
     var scanner = new Generated.Scanner(fileName, stream);
     var parser = new Generated.Parser(scanner);

     ASTNode rootNode;

     if (parser.Parse())
          rootNode = parser.Result;
}
// ...
```

Figure 4.3: Abstract Syntax Tree (Parser)

shorter or less power consuming code. Usually code optimizers rely on heuristics, since finding a good solution is a complex task. The code generator takes the IR and produces an executable target program. Here, actual memory locations and registers are assigned.

4.3 GPLEX Basics

Spark uses the Gardens Point Scanner- (GPLEX)[1] and Parser Generator (GPPG)[2]. GPLEX and GPPG take scanner and parser configuration files and produce corresponding C# implementations. The generated scanner provides either stand-alone pattern matching functionalities or may interact with a compatible parser.

Figure 4.3 shows the typical usage within a C# application. Obviously, the generated scanner and parser lie in the `Generated` name-space. First the scanner is initialized and assigned to the parser, which provides a `Parse()` method. The parser generates some kind of IR, which is in this case denoted by the ASTNode type, which is the root node of the AST.

Input Grammar

GPLEX users have to define custom behavior in a particular input file, which has to follow a certain input grammar. Basically, the input file comprises a *definition section*, a *rules section* and an optional *user code section*:

```
LexInput
   : DefinitionSection "%%" RulesSection UserCodeSection_{opt}
```

[1] `http://plas.fit.qut.edu.au/gplex/`
[2] `http://plas.fit.qut.edu.au/gppg/`

```
counter = -1;

while(counter <= 10)
{
     if(counter > 4)
          print counter;
     else
          print 0;

     counter = counter + 1;
}
```

Figure 4.4: A simplified language

```
   ;

 UserCodeSection
   : "%%" UserCode_{opt}
   ;
```

Each section is split by the `"%%"` delimiter. Most prominent definitions in the first section are:

- Name-space declaration
- Using declarations: allows the usage of types in other name-spaces
- User-code: arbitrary C# code that can be placed inside a class definition
- Lexical category definition: defines named regular expressions that can be used to build more complex expressions in the rule section

Figure 4.4 shows a simple language, which allows arithmetic expressions and also supports while- and conditional statements. The definition section of the input file could look like

```
%namespace calcInterpreter.Generated
%using calcInterpreter;
%{
    private string _fileName;
    public Scanner(string fileName, System.IO.Stream stream){
        _fileName = fileName;
        SetSource(stream);
    }
%}

DecDigit [0-9]
```

```
DecNum {DecDigit}+
IdInit [a-zA-Z]
IdChar [a-zA-Z0-9\_]
```

Here, the scanner class is placed into the `calcInterpreter.Generated` namespace. A using declaration includes the `calcInterpreter` name space. The user defined code provides a private field named `_fileName`, which corresponds to the source file. Further, a custom constructor declares that the scanner gets the character stream from a file stream. Some character class definitions conclude the definition section of the lexer configuration file:

- `DecDigit [0-9]`: digits between 0 and 9
- `DecNum {DecDigit}+`: decimal numbers as sequences one or more digits
- `IdInit [a-zA-Z]`: the first character of an identifier
- `IdChar [a-zA-Z0-9\_]`: rest of the identifier may contain digits and underscores

Having defined the above character classes, regular expressions (RE) can be declared to recognize lexemes. The rules section exhibits certain RE patterns and corresponding actions, which are triggered in case of compliance. The grammar of a rule is as follows:

```
Rule
  :  StartCondition_opt RegularExpression Action
  ;
Action
 : '|'
 | CodeLine
 | '{' CodeBlock '}'
 ;
```

The optional start condition defines a state in which the rule should be active. The rest of a rule definition is straightforward. A regular expression defines the lexeme and the action definition declares the behavior of the lexer. For each recognized character sequence the lexer returns a token type and loads up the semantic value.

The rule section for the simple language looks like:

```
"-" { return (int) '-'; }
"(" { return (int) '('; }
")" { return (int) ')'; }
"<" { return (int) '<'; }
">" { return (int) '>'; }
"=" { return (int) '='; }
"+" { return (int) '+'; }
"*" { return (int) '*'; }
```

```
"/" { return (int) '/'; }
";" { return (int) ';'; }
"{" { return (int) '{'; }
"}" { return (int) '}'; }
"." { return (int) '.'; }
">=" { return (int)Tokens.TOK_GE; }
"<=" { return (int)Tokens.TOK_LE; }
"==" { return (int)Tokens.TOK_EQEQ; }
"!=" { return (int)Tokens.TOK_NE; }
"while" { return (int)Tokens.TOK_WHILE; }
"if" { return (int)Tokens.TOK_IF; }
"else" { return (int)Tokens.TOK_ELSE; }
"print" { return (int)Tokens.TOK_PRINT; }

{DecNum}
   {yylval.number = Int32.Parse(yytext); return (int)Tokens.TOK_NUMBER;}

{IdInit}{IdChar}*
    {yylval.ident = yytext; return (int)Tokens.TOK_IDENT;}

[ \t\n]+ ; /* ignore whitespace */
```

The scanner implements an interface to interact with the parser. For each recognized lexeme the parser receives a *token type* and a *semantic value*. For every recognized decimal number, a `TOK_NUMBER` is passed. The semantic value is stored in the `yylval` field has the defined semantic value type. Here, the semantic value of a decimal number is in fact the number representation. As indicated, the `{IdInit}{IdChar}*` RE is used to match identifiers. The returned token type is `TOK_IDENT` and the semantic value is the matched character sequence. Further, white-spaces are ignored and simple operators and keywords do not return a semantic value.

4.4 GPPG Basics

The parser deals with a much more complex assignment. GPPG is an implementation of a *shift-reduce bottom-up* parser. As mentioned the parser deals with the reverse assignment presented in Section 4.2.

Again define the following rules to produce simple arithmetic expressions: `7  3 * 2+`:

```
expr : expr '+' expr (rule 1)
expr : expr '*' expr (rule 2)
expr : TOK_NUMBER    (rule 3)
```

For the expression the parser receives the following token stream:

```
TOK_NUMBER '+' TOK_NUMBER '*' TOK_NUMBER
```

Informally, the parser (depending on the configuration) performs the following steps:

```
1: .TOK_NUMBER  '+'  TOK_NUMBER  '*'  TOK_NUMBER  (shift)
2:  TOK_NUMBER .'+'  TOK_NUMBER  '*'  TOK_NUMBER  (reduce rule3)
3:  expr       .'+'  TOK_NUMBER  '*'  TOK_NUMBER  (shift)
4:  expr        '+'. TOK_NUMBER  '*'  TOK_NUMBER  (shift)
5:  expr        '+'. TOK_NUMBER  '*'  TOK_NUMBER  (shift)
6:  expr        '+'  TOK_NUMBER .'*'  TOK_NUMBER  (reduce rule3)
7:  expr        '+'  expr       .'*'  TOK_NUMBER  (shift)
8:  expr        '+'  expr        '*'. TOK_NUMBER  (shift)
9:  expr        '+'  expr        '*'  TOK_NUMBER .(reduce rule3)
10: expr        '+'  expr        '*'  expr       .(reduce rule2)
11: expr        '+'  expr                        .(reduce rule1)
12: expr                                         .(accept)
```

The dot (`.`) denotes the current position. The parser shifts if no reduction step can be applied. In step 2 the parser recognizes that `TOK_NUMBER` can be reduced to the non terminal symbol `expr`. Analogically, shift and reduction steps are performed until the start symbol is reached, which in this case is `expr`. That is, the expression follows confirms to the grammar rules and can be accepted. For this simple example already some problematic issues may be observed. In step 7 the parser could reduce the addition. This would not conform to the standard precedence rules for arithmetic expressions, since multiplication binds tighter than addition. This may be considered as a shift-reduce conflict, which introduces ambiguity for the parser. Therefore, the input file has to define according precedence rules to resolve ambiguity.

Input Grammar

Similar to the input file for GPLEX, the GPPG input files also define a particular input grammar. Again, there are three sections:

```
Grammar
    : DefinitionSequence_opt "%%" RulesSection UserCodeSection_opt
    ;
DefinitionSequence
    : DefinitionSequence_opt Declaration
    | DefinitionSequence_opt "%{ CodeBlock "%}"
    ;
UserSection
    : "%%" CodeBlock
    ;
```

The definition sequence- and user code section is optional. However, the first section comprises important definitions. Besides using- and namespace declarations also the semantic value is declared:

```
%union { public int number;
         public string ident;
         public ASTNode node; }
```

This type declaration corresponds to the semantic value the scanner initializes (e.g. `yylval.ident`, `yylval.number`). The union definition is mapped to a C# struct declaration. Furthermore, token declarations can be found in the GPPG input file:

```
Declaration
    : ... // Productions for other declarations
    | "%left" Kind_opt TokenList
    | "%token" Kind_opt TokenList
    ;

Kind
   : '<' ident '>'
   ;

TokenList
   : TokenDecl
   | TokenList ','_opt TokenDecl
   ;
TokenDecl
   : litchar
   | ident number_opt litstring_opt
   ;
```

There are different possible token declarations:

- %token: usual token declaration
- %left or %right: declares left and right associative tokens
- %nonassoc: defines tokens without assiciativity

Left and right token declarations are used to determine the binding of arithmetic operators. For example, the arithmetic expression $3 - 4 + 1$ is equivalent to $(3 - 4) + 1$, if both operators have same precedence and are left associative. If both operators are declared right associative the expression would be equivalent to $3 - (4 + 1)$, which has obviously a different meaning. Generally, left or right associativity is necessary when an operand is preceded and followed by an operator of the same precedence.

The production rule for token declarations allow an optional *kind* specification.

```
%token <ident> TOK_IDENT
```

The above definition associates the token `TOK_IDENT` with the `ident` field in the semantic value definition.

Token precedence is determined by the order in which they are declared:

```
%left '+' '-'
%left '*' '/'
```

This defines the usual arithmetic operators as left associative, giving multiplication and division a higher precedence.

The grammar of a language is defined by production rules, with non-terminal symbol on the left hand side and according right hand sides.

The parser tries to reach the non-terminal start symbol from recognized tokens, The parser uses a stack, where it pops current right hand sides and pushes back possible reductions. Therefore, also non-terminal symbols correspond to certain data structures, which can be pushed on the stack:

```
%type<node> expr stmt stmt_list program
```

The above type declaration determines that the non-terminal symbols

```
expr, stmt, stmt_list and program
```

correspond to the node field of the semantic value structure.

The (shortened but) complete definition sequence for the simple language:

```
%namespace calcInterpreter.Generated
%using calcInterpreter;
%{
    public Parser(AbstractScanner<ValueType, LexLocation> scanner)
       : base(scanner){...}

    private ASTNode _result;
    public ASTNode Result { get { return _result; } }
    private ASTNode createOperatorNode(OperatorType opType,
        params ASTNode[] operands){...}

    private ASTNode createStmtNode(StmtType stType,
        params ASTNode[] children{..}

    private ASTNode createLiteralNode(int number){...}

    private ASTNode createIdentNode(string ident){...}

    private ASTNode createStmtListNode(params ASTNode[] children){...}
%}
```

```
%union { public int number;
 public string ident;
 public ASTNode node; }

%start program

%token<number> TOK_NUMBER
%token<ident> TOK_IDENT
%token TOK_WHILE TOK_IF TOK_PRINT
%nonassoc IFX
%nonassoc TOK_ELSE

%left TOK_GE TOK_LE TOK_EQEQ TOK_NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type<node> expr stmt stmt_list program
```

Besides the discussed declarations, the user code section comprises a custom constructor definition and methods to create AST nodes.

The rules section declares the production rules. A rule definition in GPPG follows the grammar:

```
Rule
  : NonTermSymbol ':' RhsSequence_opt ';'
  ;
 RhsSequence
  : RightHandSide
  | RhsSequence '|' RightHandSide
  ;
```

basically, a non-terminal symbol serves as the left-hand side and the right-hand side may consist of optional terminal or non-terminal symbols.

The production rules of the simple language are:

```
program
: stmt_list { _result = $1; }
;
stmt_list
: stmt { $$ = $1; }
| stmt_list stmt { $$ = createStmtListNode($1, $2); }
;
```

```
stmt
: TOK_PRINT expr ';'
      { $$ = createStmtNode(StmtType.STMT_PRINT, $2); }
| TOK_IDENT '=' expr ';'
      { $$ = createStmtNode(StmtType.STMT_ASSIGN, createIdentNode($1), $3);
| TOK_WHILE '(' expr ')' stmt
      { $$ = createStmtNode(StmtType.STMT_WHILE, $3, $5); }
| TOK_IF '(' expr ')' stmt %prec IFX
      { $$ = createStmtNode(StmtType.STMT_IF, $3, $5); }
| TOK_IF '(' expr ')' stmt TOK_ELSE stmt
      { $$ = createStmtNode(StmtType.STMT_IFELSE, $3, $5, $7); }
| '{' stmt_list '}'{ $$ = $2; }
;
expr
: TOK_NUMBER { $$ = createLiteralNode($1); }
| TOK_IDENT { $$ = createIdentNode($1); }
| '-' expr %prec UMINUS
     { $$ = createOperatorNode(OperatorType.OP_UMINUS, $2); }
| expr '+' expr
     { $$ = createOperatorNode(OperatorType.OP_PLUS, $1, $3); }
| expr '-' expr
     { $$ = createOperatorNode(OperatorType.OP_MINUS, $1, $3); }
| expr '*' expr
     { $$ = createOperatorNode(OperatorType.OP_MULT, $1, $3); }
| expr '/' expr
     { $$ = createOperatorNode(OperatorType.OP_DIV, $1, $3); }
| expr '<' expr
     { $$ = createOperatorNode(OperatorType.OP_SM, $1, $3); }
| expr '>' expr
     { $$ = createOperatorNode(OperatorType.OP_GR, $1, $3); }
| expr TOK_GE expr
     { $$ = createOperatorNode(OperatorType.OP_GE, $1, $3); }
| expr TOK_LE expr
     { $$ = createOperatorNode(OperatorType.OP_LE, $1, $3); }
| expr TOK_NE expr
     { $$ = createOperatorNode(OperatorType.OP_NE, $1, $3); }
| expr TOK_EQEQ expr
     { $$ = createOperatorNode(OperatorType.OP_EQEQ, $1, $3); }
| '(' expr ')'{ $$ = $2; }
;
```

The right-hand side provides an optional semantic action, which is triggered when the parser recognizes a production rule and reduces to the symbol on the left-hand side. `TOK_NUMBER` matches the rule:

```
expr : TOK_NUMBER { $$ = createLiteralNode($1); }
     | ...
     ;
```

In the semantic action, specified in curly brackets, `$$` denotes the top of the parser stack and `$1` denotes the semantic value of the first symbol on the right-hand side (`TOK_NUMBER`).

This covers the basics of language processing required to understand the scanner and parser definition of Spark. John Gough provides a complete documentation of the input language [4], scanner- [5] and parse generator [6].

CHAPTER 5

Syntax Analysis

Foley, in his PhD thesis, describes many high-level concepts, which mostly address key ideas of the language design and system implementation. In [3, Chapter 2] and [3, Chapter 3] a lot of motivational information is provided, which led to the current application of Spark. In [3, Chapter 4] some parts of the implementation are described from a high-level point of view, which allows the reader to get a superficial stance on the system architecture. However, an in-depth documentation is missing.

In this chapter more rigorous documentation is provided, which should supplement the high-level descriptions with a more particular view on the implementation.

This chapter begins with a survey on the Spark project and describes the low-level intermediate representation which is generated by the parser.

5.1 Project Structure and System Architecture

The Spark code base comprises several sub-projects and is written in both C# and C++. Figure 5.1 shows a project dependency graph.

- *sparkc*: is written in C# and is the entry point of the compiler. Compiler options are parsed from the command line arguments and Spark shader files are fed to the compiler.
- *Spark*: is written in C# and reveals the core parts of the compiler.
- *SparkGenerateParser*: is a dummy project to generate the GPLEX scanner and GPPG parser.
- *SparkCPP*: provides a C++ interface to Spark.
- *SparkBuildLLVM*: is a dummy project to initiate the LLVM build.

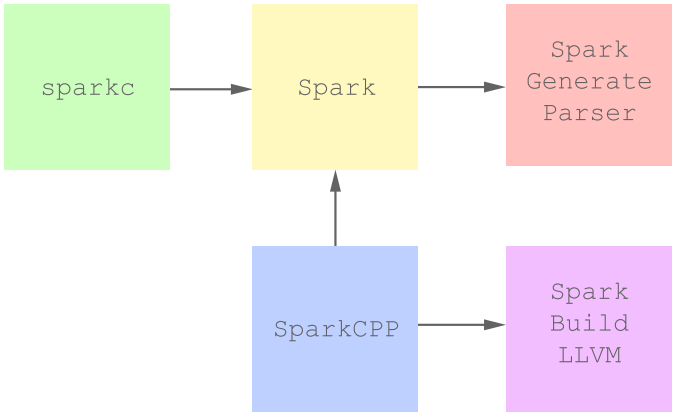

Figure 5.1: Project Dependency Graph

Compilation process

Figure 5.2 presents a system block diagram of Spark. Basically, the user specifies Spark shader classes and C++ application code. The Spark compiler, with the help of additional pipeline library interfaces, eventually, translates Spark code to HLSL or GLSL procedures. The pipeline libraries provide types, operators, functions, computation rates and plumbing operators for a specific pipeline (Direct3D 11, OpenGL 4.2).

Spark is a *source-to-source* compiler and partially conforms to the compilation procedure in Figure 4.1. Syntax analysis is followed by semantic analysis and an optimization phase (c.f. Figure 5.3). Finally, the code generator emits per-stage shader code and C++ wrapper classes.

The compiler is highly configurable via definitions in the standard library. This brings a lot benefits, however, also increases the language and compiler complexity.

Scanner and Parser

The scanner and parser specification is, of course, more complex than the examples provided in Sections 4.3 and 4.4, however the same building blocks are used. The parser defines the following semantic value structure:

```
%union { public Int64 intVal;
         public Double floatVal;
         public string stringVal;
         public AbsSourceRecord sourceRecordVal;
         public List<AbsGlobalDecl> globalDeclListVal;
         public AbsGlobalDecl globalDeclVal;
         public Identifier identifierVal;
         public List<AbsMemberDecl> memberDeclListVal;
         public AbsMemberDecl memberDeclVal;
         public List<AbsTerm> termListVal;
```

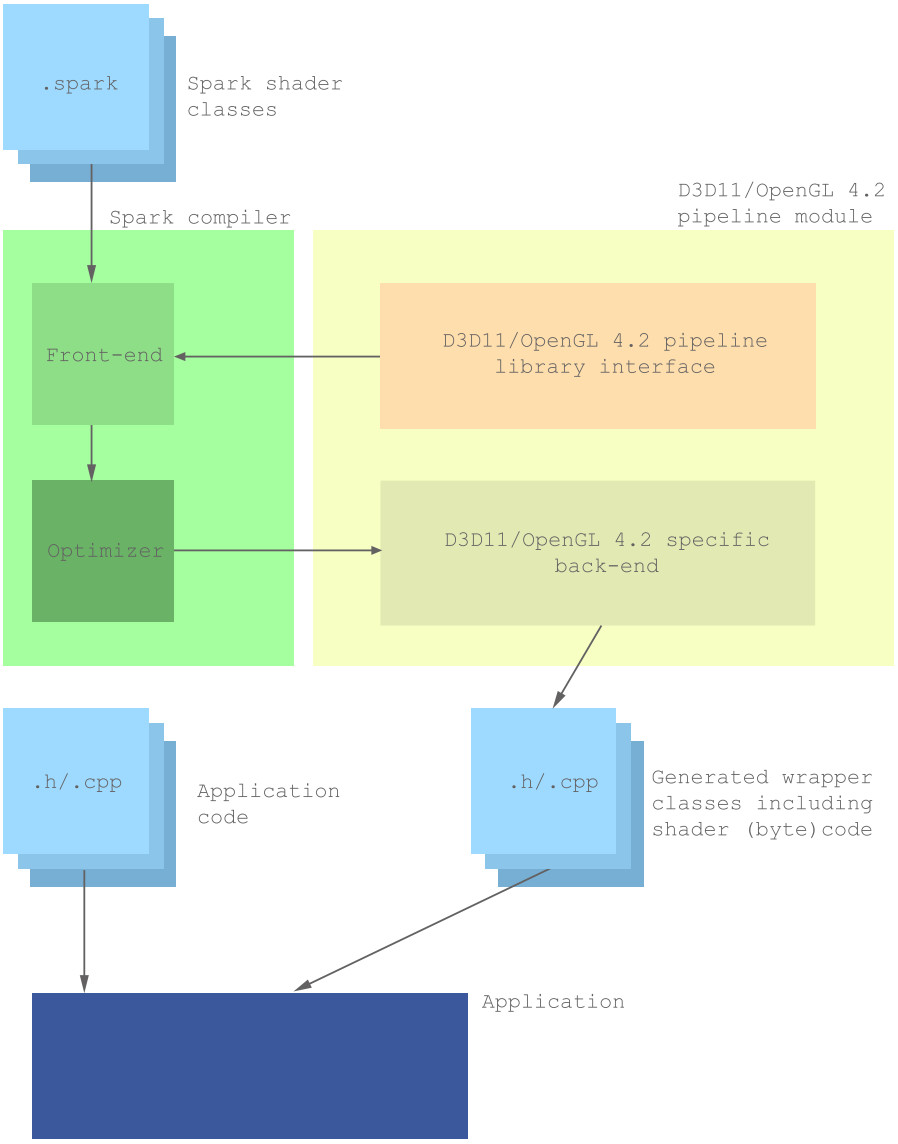

Figure 5.2: System Architecture

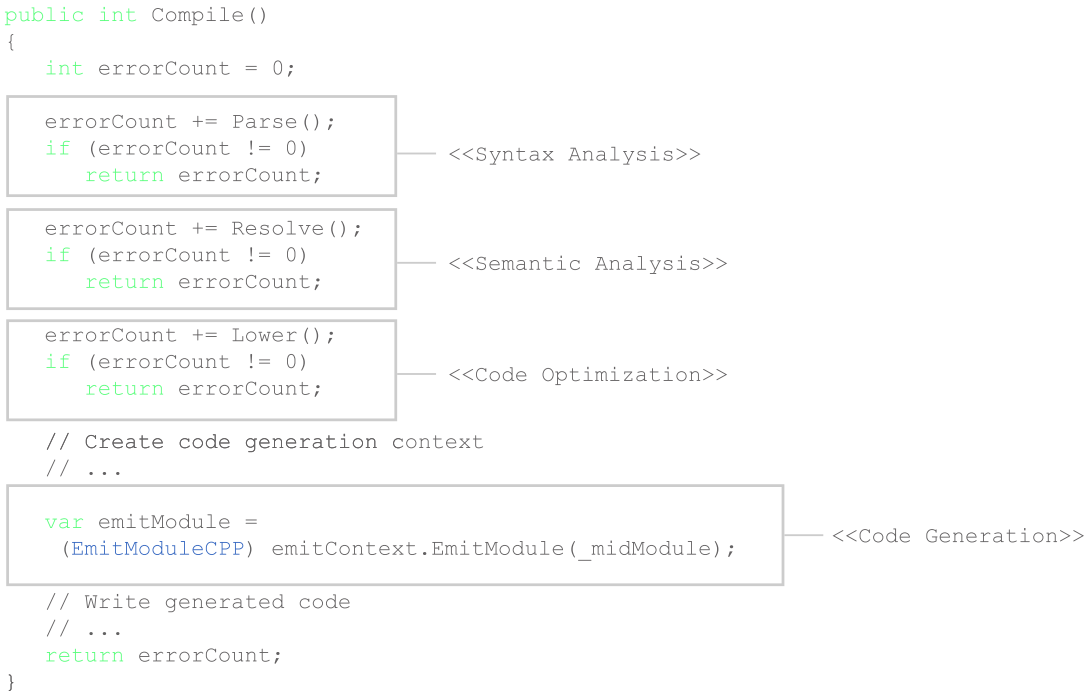

Figure 5.3: Most outer control flow of compiler (in Compiler.cs)

```
        public AbsTerm termVal;
        public List<AbsArg> argListVal;
        public AbsArg argVal;
        public List<AbsParamDecl> paramListVal;
        public AbsParamDecl paramVal;
        public List<AbsStmt> stmtListVal;
        public AbsStmt stmtVal;
        public List<AbsGenericParamDecl> genericParamListVal;
        public AbsGenericParamDecl genericParamVal;
        public AbsAttribute attributeVal;
        public AbsCase caseVal;
        public List<AbsCase> caseListVal;
        public AbsConceptDecl conceptDeclVal; }
```

Beside integer, floating point and string literal values, a list of type declarations is provided, which corresponds to the types of non-terminal symbols. Here are some of the production rules:

```
start
  : source_record
    { _result = $1; }
  ;
```

```
source_record
  : opt_global_decls
    { $$ = new AbsSourceRecord(info(@$), $1); }
  ;
opt_global_decls
  : /* empty */
    { $$ = new List<AbsGlobalDecl>(); }
  | opt_global_decls global_decl
    { $1.Add($2); $$ = $1; }
  ;
global_decl
  : pipeline_decl
    { $$ = $1; }
  ;
pipeline_decl
  : TOK_SHADER TOK_CLASS identifier opt_extends '{' opt_member_decls '}'
    { $$ = new AbsPipelineDecl(info(@$), $3, $4, $6); }
  | TOK_ABSTRACT pipeline_decl
    { $$ = $2; $$.Modifiers |= AbsModifiers.Abstract; }
  | TOK_MIXIN pipeline_decl
    { $$ = $2; $$.Modifiers |= AbsModifiers.Mixin; }
  | TOK_PRIMARY pipeline_decl
    { $$ = $2; $$.Modifiers |= AbsModifiers.Primary; }
  ;
```

A source record consists of optional global declarations. A global declaration is equivalent to a pipeline declaration. The pipeline declaration eventually is the definition of a shader so on and so forth.

5.2 Intermediate Representation

Source Record

The parser receives one or more input files and constructs an intermediate representation. Figure 5.4 shows that the parser constructs a list of *source records* (`AbsSourceRecord`). Each source record holds a list of *global declarations* (`AbsGlobalDecl`).

Pipeline Declaration

The only global declarations allowed in Spark are shader classes. Since Spark shaders may target every graphics stage, the container of a class is denoted by a *pipeline declaration* (`AbsPipelineDecl`).

A pipeline declaration has the following properties (c.f. Figure 5.5):

- Modifiers: shader modifiers, for instance, `abstract`, `mixin`, ... (`AbsModifier`)

Figure 5.4: The parser receives stdlib.spark (standard library) and custom Spark shaders (BasicSpark.spark) and produces a list of source records holding global declarations

Figure 5.5: A shader declaration with according modifiers and a class name. After the extends keyword there is a list of terms denoting the base classes. Inside the shader scope several member definitions can be found.

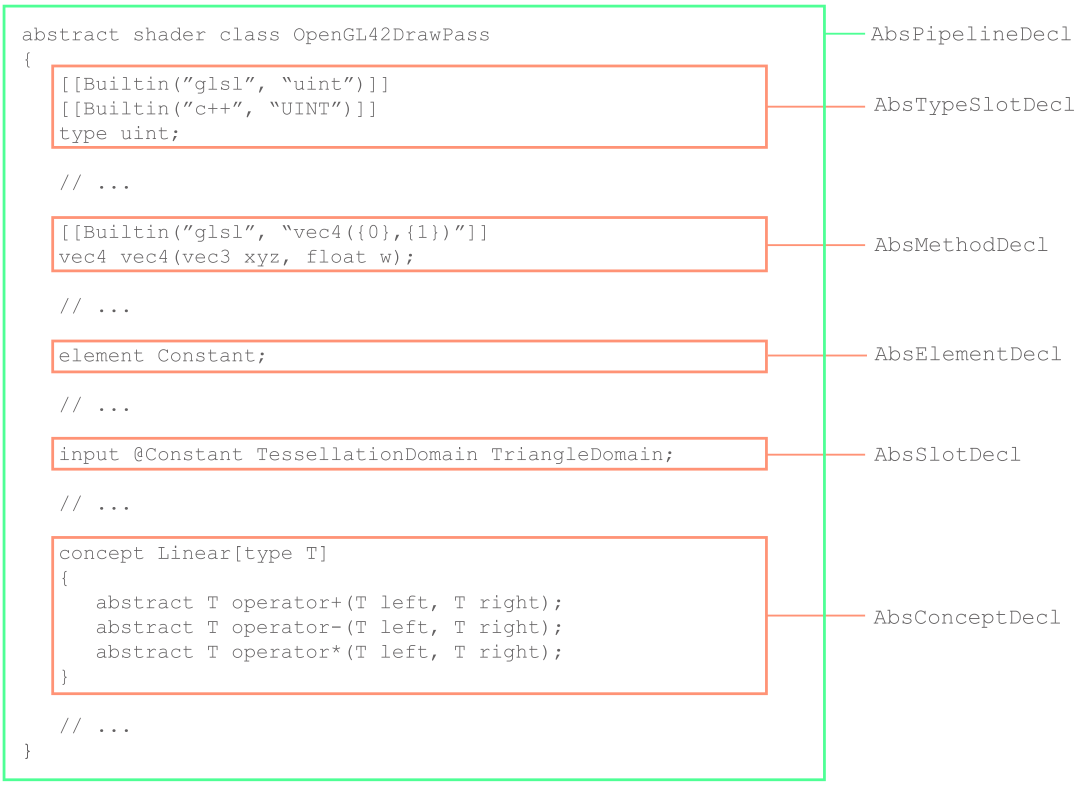

Figure 5.6: A closer look on a shader class reveals several kinds of member declarations (orange boxes)

- Identifier: the name of the class (`Identifier`)
- Bases: a list of terms denoting all base classes (`AbsTerm`)
- Members: a list of class member declarations (e.g., attributes, methods, ...) (`AbsMemberDecl`)

The data structures discussed here and in the next sections are quite complex and expose hundreds of type declarations with various derivations. For instance, `AbsTerm` is the base class of many specializations (e.g. if-term, frequency-qualified term, etc..). Whenever possible unnecessary complexity is hidden to facilitate the understanding. Actually, the `AbsTerm` instance denoting the base class can only be a variable reference (`AbsVarRef`) - as expected. Therefore, the `AbsTerm` data structure is left as a 'black box' and only currently relevant information is mentioned.

Member Declaration

Figure 5.6 shows typical member declarations in the standard library. The `OpenGL42DrawPass` declares types, attributes and methods and so forth.

Here is the list of possible member declarations:

- `AbsTypeSlotDecl`: define pipeline specific types with additional built-in tags necessary for code generation
- `AbsMethodDecl`: define methods
- `AbsElementDecl`: define record types (computation rates)
- `AbsSlotDecl`: define attributes
- `AbsStructDecl` and `AbsConceptDecl`: define nested structures and concept classes

The following properties are common to all member declarations:

- Modifiers: member modifiers (`AbsModifiers`)
- Attributes: built-in tags (`AbsAttribute`)

Type Slot Declaration

Figure 5.7 shows a type slot declaration from the standard library.

Additional built-in tags specify pipeline specific target types. The Spark *vec4* type is mapped to the glsl *vec4* and C++ vec4 type (GLM library). More precisely, each built-in attribute comprises a profile (e.g. glsl, hlsl, C++) and a template (e.g. vec4, glm::vec4).

In the lower part of Figure 5.7 a generic `Array` type is declared (e.g. Array[mat4, 10]). The array may hold arbitrary types but the size has to be specified.

There are two specializations of `AbsGenericParamDecl`.

- `AbsGenericTypeParamDecl`: used to declare a generic type
- `AbsGenericValueParamDecl`: used to declare the type of the array size parameter

The generic value parameter declaration specifies a frequency qualified term (`AbsFreqQualTerm`), to denote that the array size parameter is a per-constant integer value.

Neither HLSL nor GLSL exposes an `Array` type, therefore, the reserved `__Array` template is used provide a special code generation implementation. Chapter 8 provides more insight in emitting HLSL or GLSL code.

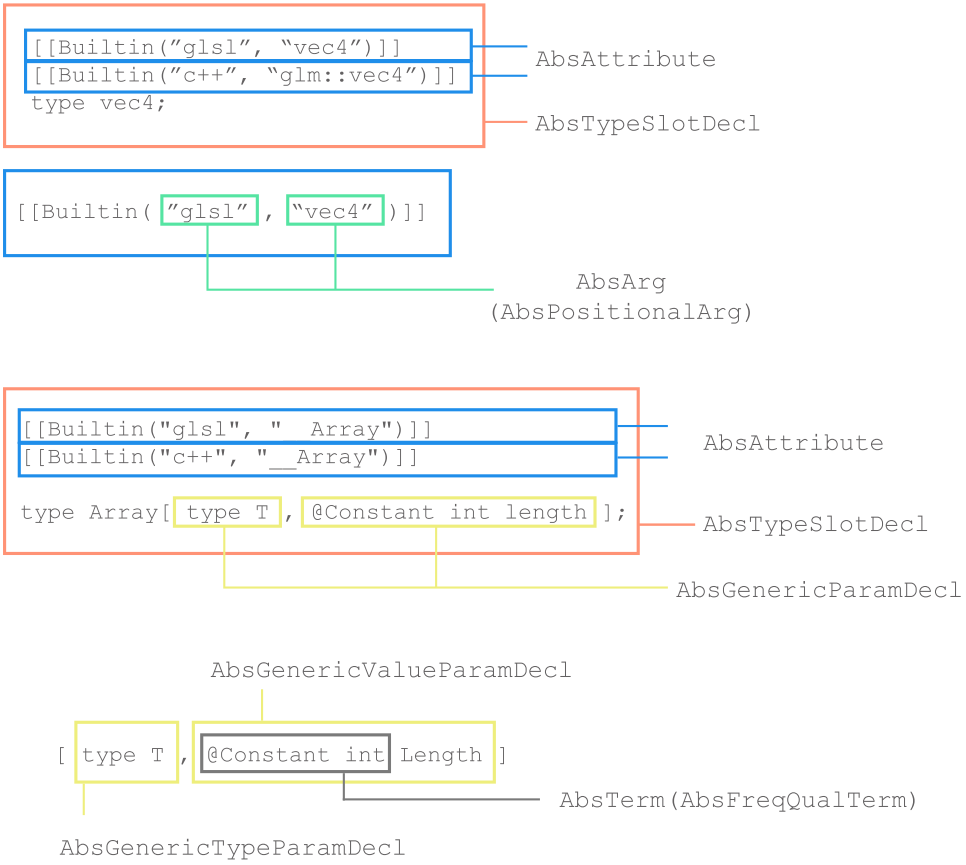

Figure 5.7: A simple type slot declaration (upper part), and a generic type declaration (lower part)

Method Declaration

Figure 5.8 shows a method definition, which is an implementation of a *plumbing operator*. `@CoarseVertex` values may be converted to `@FineVertex` values by projecting the values out of the `_c2fhelper` helper attribute. Usually, the projecting operator looks like: `value @ _c2fhelper`, however, in the standard library sometimes this notation is used: `value (_c2fhelper)`.

Method definitions expose the following properties:

- Result type: the term denoting the result type (`AbsTerm`)
- Parameters: method parameters (`AbsParam`)
- Generic parameters: generic method parameters (`AbsGenericParamDecl`)
- Method body: the implementation (`AbsStmt`)

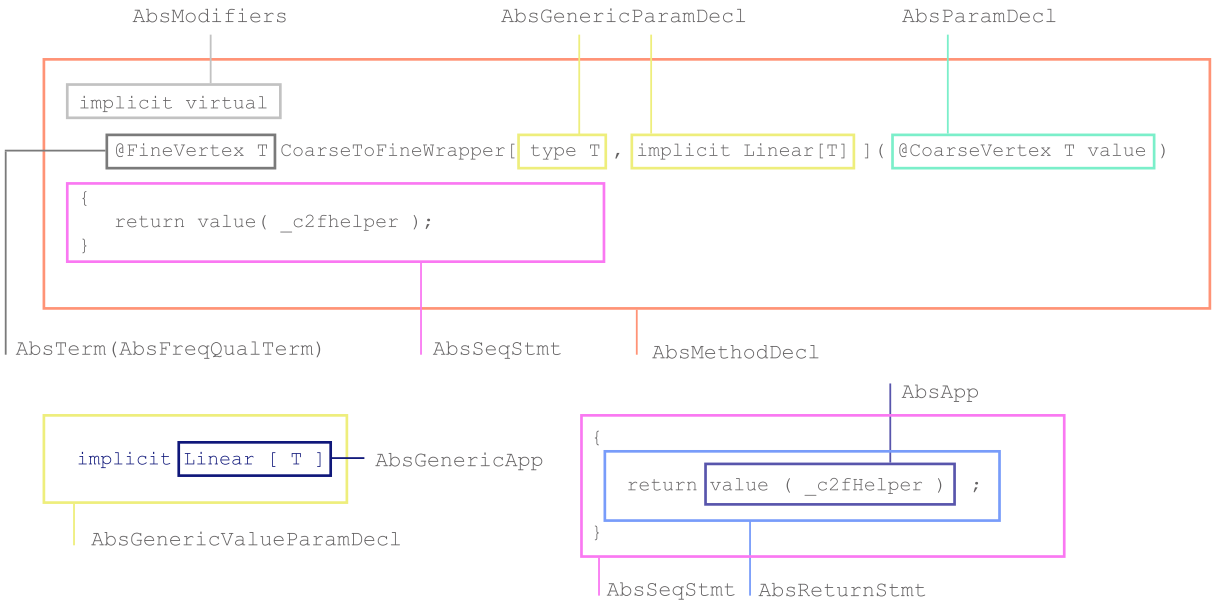

Figure 5.8: A generic method definition. A plumbing operator to convert coarse vertex- to fine vertex values.

A inherits member modifiers, which are in this case: `implicit` and `virtual`. That is, the plumbing operator may be implicitly inserted by the compiler whenever a conversion is required. Further, the conversion operation may be overridden by derived classes.

The generic parameter declaration enforces the `Linear` concept to be supported. The linear concept requires that the addition, subtraction and multiplication is defined. This matches the usual interpolation requirements for attributes.

The real interesting part are sequence of statements in the method body.

Attribute Declaration

Attribute declarations (`AbsSlotDecl`) expose the following properties:

- Type: a term denoting the attribute type (`AbsTerm`)
- Initialization: the initialization term (`AbsTerm`)
- Input flag: mark as input attribute (`AbsModifier`)

Figure 5.9 shows a common slot declaration. The normal matrix is computed as the transpose inverse model-view matrix. Each nested method call is represented as a method application (`AbsApp`).

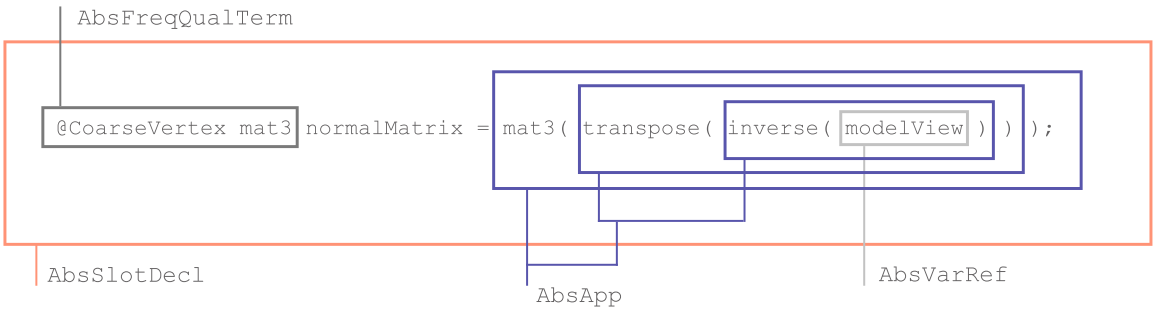

Figure 5.9: Attribute definition (AbsSlotDecl)

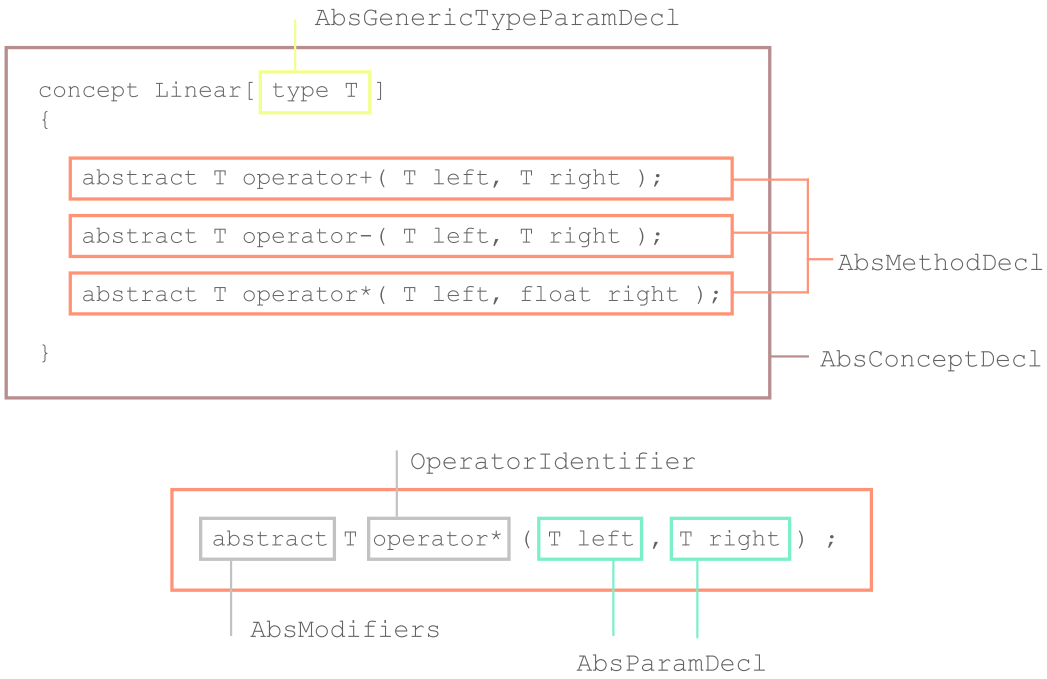

Figure 5.10: Linear concept declaration

Concept Declaration

Figure 5.10 shows a generic concept class declaration consisting three method signatures, which must be implemented to support the linear concept.

The IR, which is delivered by the parser has been discussed to some extend. Further details are introduced as necessary. In the next chapter some semantic analysis tasks are discussed.

CHAPTER 6

Semantic Analysis

In [3, Chapter 2 and Chapter 3] Foley provides lots of conceptual insights into many aspects of the Spark shading language. However, a specific view on the implementation is not provided. Here, both the semantic validation of those concepts and the process itself is documented in more detail. Some aspects of the intermediate representation (IR) of the parser have been provided in Chapter 5. As a next step, semantic analysis (resolve phase) is performed. Basically, each step of the compiler takes an IR and constructs a new enhanced representation. This chapter discusses the new data structures, the build system and some semantic analysis tasks.

6.1 Build System And Lazy initialization

IR generation and semantic analysis is implemented via a lazy initialization mechanism. Figure 5.3 shows the initiation of semantic analysis by calling the `Resolve` method of the compiler. A *resolve context* is created, which implements the crucial semantic analysis tasks.

Essential Interfaces

There are five central interfaces which provide the basic facilities for the lazy initialization mechanism.

Figure 6.1 shows the relationship between those interfaces. The basic `ILazy` interface declares merely a `Force()` method signature. The more refined `ILazy<out T>` interface appends a generic value property. That is, the `ILazy<out T>` implementations contain a generic value property, which is initialized when the actual instance is needed. The `ILazyFactory` interface exposes a method to add and maintain `Ilazy` instances. The `INewBuilder` interface extends the `ILazy` interface by allowing to add *build actions* for certain *build phases*. The The abstract class `NewBuilder<T>` extends both `ILazy<T>` and `INewBuilder`.

```
public interface ILazy
{
   void Force();
}

public interface ILazy<out T> : ILazy
{
   T Value { get; }
}

public interface ILazyFactory
{
   void Add(ILazy instance);
}

public interface INewBuilder : ILazy
{
   void AddAction(NewBuilderPhase phase, Action action);
}

public abstract class NewBuilder<T> : ILazy<T>, INewBuilder
   where T : class { // ... }
```

Figure 6.1: Essential interfaces for the lazy initialization mechanism (in Builder.cs)

Build System

The `NewBuilderPhase` defines seven build phases: *Initial*, *Dependencies*, *Header*, *Body*, *Seal*, *Finalize* and *Final*. Children of `NewBuilder<T>` may specify build actions via C# lambda expressions. The `LazyFactory` class implements the `ILazyFactory` interface and maintains `ILazy` instances. That is, either `ILazy` implementations or `NewBuilder<T>` implementations may be added to the `LazyFactory`. The implementation of the `Force()` method of the `ILazy` interface is simple: each `Lazy` instance is associated with a generator function (again specified as a C# lambda expression), which is called when the actual value of the Lazy instance is enforced. Actual values of the children of the abstract class `NewBuilder<T>` are computed by applying each build action of each build phase. The builder keeps track of the phases where build actions are already added and phases where the build actions are already applied. Build actions, which are added or performed too late or too early may be caught that way.

Figure 6.2 shows the inner loop of the resolve phase. While there are uninitialized lazy instances, the actual values are enforced. Each lazy instance may trigger new lazy instances to be added to the factory. Therefore the resolve method iteratively resolves all lazy instances, until there are no unresolved instances.

In the first iteration the entire module declaration, which is the most outer container, is added to the lazy factory. The module declaration comprises global declarations, which also have to be resolved. The build actions, which need to be performed are different from task to task. Trivial building steps may merely require a conversion from a `List<T>` to `T[]`. More complex build

```
public class LazyFactory : ILazyFactory
{
    // ...

    public void Force()
    {
        while (true)
        {
            if (_instances.Count == 0)
                return;

            var oldInstances = _instances;
            _instances = new List<ILazy>();

            foreach (var instance in oldInstances)
                instance.Force();
        }
    }

    public void Add(ILazy instance){ _instances.Add(instance); }

    private List<ILazy> _instances = new List<ILazy>();
}
```

Figure 6.2: Inner loop of the resolve phase (in Builder.cs)

```
public interface IResModuleDecl
{
    IEnumerable<IResGlobalDecl> LookupDecls(Identifier name);
    IEnumerable<IResGlobalDecl> Decls { get; }
}
```

Figure 6.3: IResModuldeDecl interface (in IResModuleDecl.cs)

tasks defer new lazy instances and are mostly specified using C# lambda expressions. In the next subsection more details about the initialization mechanism and IR generation is provided.

6.2 Module Declaration

The module declaration is the root data structure holding all global declarations, as delineated in Figure 6.3.

Figure 6.4 shows a typical code fragment during semantic analysis. At the beginning, the resolve method of the previously mentioned resolve context builds a resolved module declaration (`ResModuleDecl`) with the help of the corresponding resolve module declaration builder (`ResModuleDeclBuilder`). The `ResModuleDeclBuilder` derives from:

```
public IResModuleDecl Resolve(IEnumerable<AbsSourceRecord> sourceRecords)
{
   var lazyResModule = ResModuleDeclBuilder.Build(
      LazyFactory,
      (resModuleBuilder) =>
      {
         var globalScope = SetupBuiltins(resModuleBuilder);
         var moduleScope = new ResModuleScope(resModuleBuilder);

         var env = new ResEnv(this, _diagnostics, globalScope);
         env = env.NestScope(moduleScope);

         foreach (var sr in sourceRecords)
            foreach (var decl in sr.decls)
               ResolveGlobalDecl(resModuleBuilder, decl, env);
      });

   _lazyFactory.Force();
   return lazyResModule.Value;
}
```

Figure 6.4: For the list of source records, create a module declaration, where each global declaration is resolved (in ResolveContext.cs)

`NewBuilder<IResModuleDecl>`

As described in Section 6.1, the `ResModuleDeclBuilder` allows the association of build actions with build phases. Children of `NewBuilder<IResModuldeDecl>` are, due to emulation of *multiple inheritance* through interfaces, also children of `ILazy<IResModuleDecl>`. The key idea is that the `ResModuldeDecl` instance is the *value property* of the lazy instance. That is, to initialize the value of the lazy instance all build actions have to be performed for each build phase. The `ResModuldeDecl` class, more precisely, implements the `IResModuleDecl` interface, which simply exhibits resolved global declarations and a method to lookup global declaration for a given identifier.

Since global declarations are not initialized yet, an `ILazy<IResGlobalDecl[]>` instance is bound to the module declaration.

The build action (header phase) of the `ResModuldeDeclBuilder` is given by the lambda expression (c.f. Figure 6.4), where semantic analysis of the global declarations is deferred. The anonymous method adds built-in types (bool, int, float) in the global scope. The module declaration is nested inside the global scope. The initialization of the `IResGlobalDecl[]` instance is associated with the simple generator function:

```
() => builder._decls.ToArray()
```

Since the initialization of the global declarations depend on the initialization of the module declaration the lambda expression is further nested in:

```
public interface IResPipelineDecl : IResGlobalDecl
{
    IResVarDecl ThisParameter { get; }

    IEnumerable<IResMemberNameGroup> LookupMembers(Identifier name);
    IResMemberLineDecl FindMember(IResMemberSpec memberSpec);

    IResFacetDecl DirectFacet { get; }
    IEnumerable<IResFacetDecl> Facets { get; }
    IEnumerable<IResMemberDecl> Members { get; }

    IEnumerable<IResMemberDecl> ImplicitMembers { get; }

    ResMixinMode MixinMode { get; }
    ResMemberConcretenessMode ConcretenessMode { get; }
}
```

Figure 6.5: IResPipelineDecl interface (in IResPipelineDecl.cs)

```
() => { Force(phase); return generator(); }
```

The nested lambda expression enforces that all actions are flushed for the module declaration to generate the the global declarations afterwards. The same initialization patters reoccurs through the front- and back-end of the Spark compiler. In the next iteration of the initialization the added lazy instances may append other instances, analogically. The next sections describe the initialization of global- (pipeline-), facet- and member declarations.

6.3 Pipeline Declaration

A module declaration holds global declarations, that is pipeline declarations. Resolving the shader declarations is the main target of the Spark compiler.

Before describing the detailed initialization process, first examine the `IResPipelineDecl` contract, which an actual resolved pipeline declaration implements (Figure 6.5). Classes implementing the interface exhibit, *mixin-* and *concreteness* modifiers of the shader class and the concept of a *facet declaration*. The facet declaration wraps the key aspects of a shader class and is described in 6.4. Further, for given member specifications the corresponding member line declaration may be found. It is not necessary to understand every interface and data structure immediately, since each initialization is discussed separately.

Similar to Figure 6.4 in Figure 6.6 a lambda expression specifies a method call to `BuildResolvedShaderClassDecl`, which resolves the given shader class declaration. In the lower part of Figure 6.6 the construction of necessary new lazy instances is indicated by the grey rectangle. Simple parameters like `_thisParamter`, `_concretenessMode` or `_mixinMode` do not require complex build actions. Therefore, the lambda expression, which specifies the generation function, just returns the parameter itself. Next, the actual initialization process `BuildResolvedShaderClassDecl(..)` is discussed.

```
private void ResolveShaderClassDecl(
   ResModuleDeclBuilder resModule, ResEnv env, SourceRange range, Identifier name,
   AbsModifiers modifiers, Func<ResEnv, IResPipelineRef[]> resolveBases,
   IEnumerable<AbsMemberDecl> absMembers, bool ignoreOrderingErrors )
{
   var resPipeline = ResPipelineDecl.Build(
      LazyFactory, resModule, range, name,
      (resShaderClassBuilder) =>
      {
         BuildResolvedShaderClassDecl(
            resShaderClassBuilder, resModule, env, range, name, modifiers,
            resolveBases, absMembers, ignoreOrderingErrors );
      });
   resModule.AddDecl(resPipeline);
}
```

```
public ResPipelineDeclBuilder(
   ILazyFactory lazyFactory, ILazy<IResModuleDecl> module, SourceRange range,
   Identifier name )
      : base(lazyFactory)
{
   _module = module;
   _range = range;
   _name = name;

   var resShaderClass = new ResPipelineDecl(
      _module, _range, _name,
      NewLazy(() => _thisParameter),
      NewLazy(() => _directFacet.Value),
      NewLazy(() => (from f in _facets select f.Value).Eager()),   <<new lazy instances>>
      NewLazy(() => _mixinMode),
      NewLazy(() => _concretenessMode));

   SetValue(resShaderClass);
}
```

Figure 6.6: Resolve Pipeline Declaration (upper part in ResolveContext.cs, lower part in ResPipelineDecl.cs)

Class Modifiers

A shader class is either partially implemented and, therefore abstract, or concrete. Further, a class is either declared primary or mixin. Semantic analysis has to confirm that a class is not both mixin and primary.

Resolving Base Class References

A shader class may derive from multiple mixin classes but only from a single primary class, which has to be listed first in the list of bases. Semantic analysis has to confirm that the identifiers in the inheritance list of a shader class declaration reference shader class declarations. This is achieved in two steps. First, the resolve environment and scoping is used to ensure that the identifier references a class declaration. Then the compiler implicitly coerces the reference

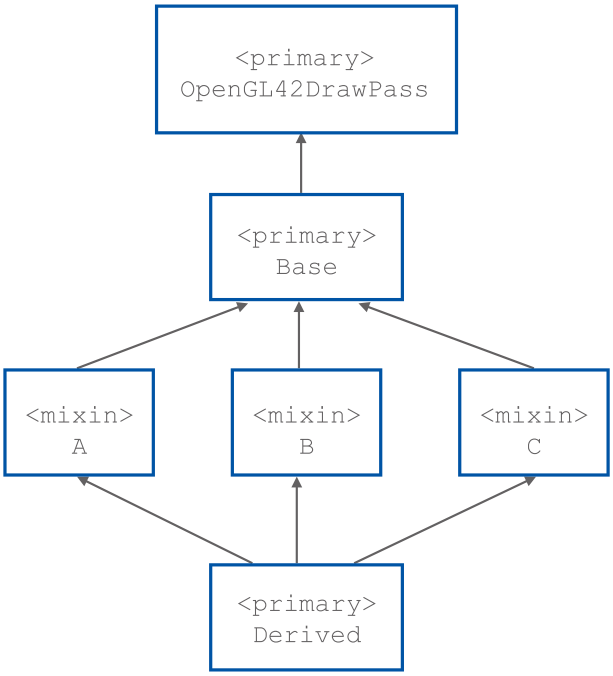

Figure 6.7: Diamond pattern multiple inheritance problem

expression to a pipeline reference. Next the inheritance hierarchy is flattened according to the C3 linearization algorithm. While constructing the pipeline declaration also the facets build actions are deferred to the lazy factory. First the derived facets are created in the correct order, then the direct facet can be linked to the inherited facets. Then all member declarations are resolved.

Base Linearization

For the given primary class `Derived` the bases are flattened to avoid the problems of multiple inheritance. Figure 6.7 shows the problematic structure. The user defined class `Derived` derives from both `A`, `B` and `C`. Those classes derive from the single `Base` class. What if in `Derived` a method from `Base` is called, which is overridden in `A`, `B` and `C`? To avoid ambiguity the C3 linearization algorithm is implemented. In Spark multiple inheritance is supported by allowing sub-classes to derive from a single primary class and multiple mixin classes.

For the class declaration

```
abstract shader class Derived extends A, B, C { //... }
```

the bases comprise the following facets:

- `A` derives from `Base`, so has the direct facet `A` and inherited facets `Base` and `OpenGL42DrawPass`: {A, Base, OpenGL42DrawPass}

- `B` derives from `Base`, so has the direct facet `A` and inherited facets `Base` and `OpenGL42DrawPass`: {B, Base, OpenGL42DrawPass}
- `C` derives from `Base`, so has the direct facet `A` and inherited facets `Base` and `OpenGL42DrawPass`: {C, Base, OpenGL42DrawPass}

The linearization algorithm flattens the inheritance hierarchy in reverse order. So, basically,

$$\{\text{C, Base, OpenGL42DrawPass}\}, \{\text{B, Base, OpenGL42DrawPass}\}, \{\text{A, Base, OpenGL42DrawPass}\}$$

has to be linearized.

First {C, Base, OpenGL42DrawPass} and {B, Base, OpenGL42DrawPass} is linearized.

$$Linearize(\{\text{C, Base, OpenGL42DrawPass}\}, \{\text{B, Base, OpenGL42DrawPass}\})$$

The algorithm recursively checks whether the most left facet from the left set is already in the right facet. If the right set contains the current element, it can be removed from the left set. Otherwise the flattened inheritance consists of the most left element and the linearization of the rest. That is:

$$\{C\} \bigcup Linearize(\{\text{Base, OpenGL42DrawPass}\}, \{\text{B, Base, OpenGL42DrawPass}\})$$

$$\{C\} \bigcup Linearize(\{\text{OpenGL42DrawPass}\}, \{\text{B, Base, OpenGL42DrawPass}\})$$

$$\{C\} \bigcup Linearize(\{\}, \{\text{B, Base, OpenGL42DrawPass}\})$$

$$\{C\} \bigcup \{\text{B, Base, OpenGL42DrawPass}\}$$

$$\{\text{C, B, Base, OpenGL42DrawPass}\}$$

Next the other list is linearized:

$$Linearize(\{\text{C, B, Base, OpenGL42DrawPass}\}, \{\text{A, Base, OpenGL42DrawPass}\})$$

$$\{C\} \bigcup Linearize(\{\text{B, Base, OpenGL42DrawPass}\}, \{\text{A, Base, OpenGL42DrawPass}\})$$

$$\{C, B\} \bigcup Linearize(\{\text{Base, OpenGL42DrawPass}\}, \{\text{A, Base, OpenGL42DrawPass}\})$$

$$\{C, B\} \bigcup Linearize(\{\text{OpenGL42DrawPass}\}, \{\text{A, Base, OpenGL42DrawPass}\})$$

$$\{C, B\} \bigcup Linearize(\{\}, \{\text{A, Base, OpenGL42DrawPass}\})$$

$$\{C, B\} \bigcup \{\text{A, Base, OpenGL42DrawPass}\}$$

Finally, the flattened inheritance order is

$$\{\text{C, B, A, Base, OpenGL42DrawPass}\}$$

The list is ordered from left to right containing most derived bases on the left and most general bases on the right.

Populating Base Shader Facets

For the computed linearization, facet declarations are created and added to the facet collection of the current shader declaration, unless the according facet declaration already exists. Further, the facet hierarchy is interlinked, that is for inherited facet declarations the base facets are added.

Populating Member Line Declarations

The next step is to iterate through the linearization of the base classes and populate the member declarations to the current shader class. The member declarations of each base facet declaration are bound to the *this* parameter of the current shader class. For direct member declarations additional information is collected. With the assembled facet information the member line declarations are initialized. The linearized bases are used again to append inherited member line declarations to the derived facet declarations.

Resolving Member Declarations

After establishing the inherited shader facet declarations with all member declarations populated for the inherited facets, now, all member declarations have to be resolved. First, all direct member declarations are resolved, then the current shader class is sealed, and then the override member declarations are considered. The structure of member declarations and the according semantic analysis is discussed in Section 6.5.

Final Checks

Finally, a lambda expression is attached to the pipeline declaration, which performs the following checks:

- A non-abstract class must not contain abstract member declarations

- For all inherited shader facet declarations it is ensured that, if there are more inherited primary shader classes, that these other primary classes are super classes of the actual primary base class of the current shader class.

6.4 Facet Declaration

Beside modifiers and concreteness modes, a resolved shader class declaration exhibits a facet concept. A facet comprises members declarations, which are the decisive parts of a class declaration. Therefore, a class has at least a direct facet declaration with its own class members and inherited facets. In the architecture setup of Spark the Direct3D and OpenGL base classes are the only classes, which comprise only a direct facet declaration. Through facet declarations the class declaration exhibits also member declarations. The facet declaration keeps track of facet declarations of the direct bases.

As mentioned a shader class most prominently maintains direct and indirect inherited shader facets. After finding a linearization of the inheritance tree, base facets are created for the current shader class according to the ordering determined by the linearization. Therefore, the `Facets` property of a resolved class declaration will contain its direct declaration as the first element and then most derived to most general facets. In a next step also the direct bases are added. Finally, all inherited members are added.

6.5 Member Declaration

A great part of semantic analysis addresses member declarations. Again, a module declaration holds, possibly, several pipeline declarations. Each shader may exhibit computations, which may target the whole graphics pipeline. The core aspects of a pipeline declaration is wrapped by facet declarations. Each facet declaration holds member declarations. In the following subsections the semantic analysis of different kinds of member declarations are described. However, note the relationship between the `IResMemberDecl` and `IResMemberLineDecl` interface in Figure 6.8.

Basically, core aspects of a member declaration are wrapped by the `IResMemberLineDecl` interface, which is nested in the `IResMemberDecl` interface. All concrete member declarations at least provide an effective declaration and inherited declarations. Characteristically, each member declaration is assigned a *lexical id* and augmented by certain member declaration tags, which are used during code generation. Concreteness and declaration mode attributes determine whether the member declaration is abstract, virtual or final and inherited, direct or extended, respectively.

Type Slot Declaration

For a type slot declaration there are actually two major distinctions.

```
[[Builtin("glsl", "ubyte")]]
[[Builtin("c++", "char")]]
```

```
public interface IResMemberDecl
{
    SourceRange Range { get; }
    Identifier Name { get; }
    IResMemberLineDecl Line { get; }
    IResMemberRef MakeRef(SourceRange range, IResMemberTerm memberTerm);
}

public interface IResMemberLineDecl
{
    Identifier Name { get; }
    ResLexicalID OriginalLexicalID { get; }
    ResMemberCategory Category { get; }
    IResMemberDecl EffectiveDecl { get; }
    IEnumerable<IResMemberDecl> InheritedDecls { get; }

    ResMemberConcretenessMode ConcretenessMode { get; }
    ResMemberDeclMode MemberDeclMode { get; }
    IEnumerable<ResTag> Tags { get; }
}
```

Figure 6.8: IResMemberDecl (in IResMemberDecl.cs), IResMemberLineDecl (in IResPipelineDecl.cs)

```
type ubyte;
```

For the above case of a simple type declaration, semantic analysis has to check that overriding and overloading is not allowed. Further, the tags are resolved and adapted to the resolved intermediate representation (c.f. 6.5).

```
[[Builtin("hlsl", "__Array")]]
[[Builtin("c++", "__Array")]]
[[Builtin("llvm", "__Array")]]
type Array[type T, @Constant int Length];
```

For the above generic type declaration, additionally, the generic parameters are resolved, which is described in 6.5.

Element Declaration

```
element Uniform;
```

An element declaration is a record type declaration. Similar to type slot declarations overriding is not allowed. There is no semantic analysis performed for element declarations.

```
public interface IResAttributeDecl : IResMemberDecl
{
    IResFreqQualType Type { get; }
    IResExp Init { get; }
    ResAttributeFlags Flags { get; }
}
```

Figure 6.9: IResAttributeDecl (in IResAttributeDecl.cs)

```
var candidates = (from facet in resContainer.ContainerDecl.InheritedFacets
                  let mng = facet.FindMemberNameGroup(absAttribute.Name)
                  where mng != null
                  let mcg = mng.FindMemberCategoryGroup(category)
                  where mcg != null
                  from ml in mcg.Lines
                  let decl = ml.EffectiveDecl
                  where IsAttributeOverloadMatch(
                    ml,
                    decl,
                    absAttribute,
                    matchType,
                    env)
                  select ml).ToArray();
```

Figure 6.10: C# LINQ expression used to search for overload matches (in ResolveContext.cs)

Slot Declaration

Characteristically, most computations are either specified as frequency qualified attribute- or method declarations.

Figure 6.9 shows the contract, which is implemented by an actual attribute declaration. Attribute declarations may exhibit attribute flags: *None*, *Optional*, *Input* or *Output*. Optionally, the attribute declaration may be initialized by some kind of expression. Consequently, semantic analysis targets the resolving and type checking of the frequency qualified type- and the initialization expression. Besides, the attribute- and member declaration flags are checked. In Spark an attribute declaration may be declared virtual or abstract.

```
abstract output @Uniform ubyte dummyVar;
```

For the above attribute declaration semantic analysis performs the following steps: First the frequency qualified term is coerced to a `IResFreqQualType` as discussed in 6.6. Overridden attribute declarations do not have a resolved member line builder attached. Therefore, the C# LINQ expression in 6.10 iterates through the inherited facets to find the according member name- and category group. If there exists such a resolved category group the available member line declarations are checked for matching declarations.

The candidate is decided to be matching when the following attribute pattern is found:

```
public interface IResGenericDecl : IResMemberDecl
{
    IEnumerable<IResGenericParamDecl> Parameters { get; }
    IResMemberDecl InnerDecl { get; }
}
```

Figure 6.11: IResGenericDecl interface (in IResGenericDecl.cs)

- `MemberDeclMode` is forced to be `ResMemberDeclMode.Inherited`
- `ConcretenessMode` is forced to be either abstract or virtual
- Type expressions of the actual attribute- and candidate declaration have to match

The type expressions have to match exactly or are enforced to be some kind of sub-type of each other according to the type system of Spark [3, Section 2.3.4]. If multiple candidates are found the overridden attribute declaration is ambiguous and error message is generated. If no candidates can be found, no matching declaration is in the available scope. In that case an artificial member line declaration is constructed. Additionally, semantic analysis confirms that *input* variables cannot be overridden and *non-optional* attributes, when overridden, cannot be re-defined as optional. The according `ResAttributeDeclBuilder` is constructed and a build action, which provides semantic analysis for the initialization expression, is appended. Final checks confirm that abstract and input declaration cannot have an initialization. Also, non-abstract, non-input attributes must be defined. The term specifying the method body is resolved (c.f. Section 6.6) and coerced to the type expression of the attribute declaration.

Method Declaration

There are two types of method declarations. More complex generic method declarations require the resolving and embedding of the generic parameter declarations as described in Subsection 6.5. For non-generic method declarations the method parameters are resolved and nested into the method scope (c.f. Section 6.5). The `ResMethodDecl` instance is initialized for non-generic method declaration, otherwise additionally a `ResGenericDecl` instance is created.

Figure 6.11 shows the `IResGenericDecl` interface, which extends the `IResMemberDecl` interface and holds an inner member declaration. In this case, the generic declaration holds the actual method declaration. For overriding method declarations the virtual or abstract method declaration has to be found. Similar to attribute overriding in Subsection 6.5 possible method declaration candidates are found using a C# LINQ expression. The build action appended to the `ResMethodDeclBuilder` performs semantic analysis on the possible member line tags and result type (c.f. Section 6.6) of the method declaration. If the result type is specified by a frequency-qualified type expression, all input parameters must also be frequency-qualified types, and vice versa. Finally, the method body is resolved (c.f. Section 6.7). Method declarations are categorized in three different classes. *Ordinary* method declaration have simple type expressions, whereas *SingleFreq* method declarations provide frequency-qualified return and parame-

ter types. If the frequency-qualified type of the return value differs from the frequency-qualified input parameters the method is classified as *Conversion*.

Concept Class Declaration

```
concept Linear[type T]
{
    abstract T operator+(T left, T right);
    abstract T operator-(T left, T right);
    abstract T operator*(T left, float right);
}
```

For the above *linear concept* the semantic analysis has to confirm that neither overriding nor overloading is allowed. Further, possible generic parameters are resolved and nested in a generic parameter scope as discussed in Section6.5. Characteristically, the `ResConceptClassBuilder` is used to initialize the actual `ResConceptClassDecl` via a build action specified as a C# lambda expression. Basically, the anonymous method resolves all member declarations of the concept class declaration (c.f. Section 6.5). For generic parameterized concept class declarations a `ResGenericDecl` instance is created, which holds the actual resolved concept class declaration.

Struct Declaration

```
struct PNU
{
    vec3 position;
    vec3 normal;
    vec2 texCoord;
}
```

The semantic analysis implementation for a *structure declaration* simply initializes the `ResStructDecl` instance via `ResStructDeclBuilder` by specifying an anonymous method, which resolves the inner member declarations.

Field Declaration

Actually, the members of the structure declaration in Section6.5 are recognized as *field declarations*. The build action, which is used to initialize the `ResFieldDecl` instance, resolves the type expression of the field. If the field is initialized the term on the right hand side is resolved (c.f. Section 6.6) and coerced to the actual implicit computation rate of the current environment.

Member Declaration Tag

When resolving tags only the *built-in* meta-tag is recognized. A tag comprises a *profile* and *template* property. Built-in tags are discussed in more detail in Chapter 8.

Parameter Declaration

An abstract parameter declaration is resolved by initializing a `ResVarDecl` instance and appending a build action, which resolves the type expression of the parameter declaration. The term specifying the type expression is first resolved and then coerced to be an actual type expression.

Generic Parameter Declaration

There are two types of abstract generic parameter declarations.

- `AbsGenericTypeParamDecl`
- `AbsGenericValueParamDecl`

An abstract generic type parameter declaration is simply initialized and kinded as a proper type and inserted in the current parameter scope. For abstract generic value parameters a resolved variable declaration is initialized and inserted into the current scope. The build action enforces the type parameter to be resolved and coerced to a proper type or a frequency-qualified type of proper types.

6.6 Resolving Terms

Frequency-Qualified Types

The resolving of frequency-qualified terms involves the resolving of the frequency- and type expression. The expression specifying the frequency type must reference an element declaration and is resolved as discussed in Subsection 6.6 and coerced to an element reference.

Variable References

A variable reference is looked-up in the current scope querying the according *member name*- and *category groups*. Due to overriding there might be more than a single implementation, which also has to be resolved for the current context. If the variable reference cannot be found, an error message is emitted.

Method Application

A method application is initialized by resolving the abstract term, which denotes the function and the application parameters. Usually, the function term is a variable reference to a method declaration and is resolved as discussed in Section 6.6. Eventually, the function arguments have to be resolved. For the resolved application reference a matching candidate has to be found, which involves the following semantic analysis tasks:

First candidates are created and eventually the candidates are filtered to find the actual matching method application.

```
private interface IResCandidate<AbsAppT>
{
    IResCandidate<AbsAppT> Filter(
        Func<ResCandidate<AbsAppT>, bool> filter);

    IEnumerable<ResCandidate<AbsAppT>> AllCandidates { get; }

    ResCandidate<AbsAppT> BestCandidate { get; }
}
private abstract class ResCandidate<AbsAppT> : IResCandidate<AbsAppT>
{
    public ResolveContext _context { get; set; }
    public DiagnosticSink _diagnostics { get; set; }
    public ResEnv _env { get; set; }
    public ResArg<IResTerm>[] _rawArgs { get; set; }
    public SourceRange _range { get; set; }
    public IResGenericArg[] _dummyArgs { get; set; }

    public abstract bool CheckArity();
    public abstract bool CheckTypes();
    public abstract IResTerm Gen();

    public IResCandidate<AbsAppT> Filter(
        Func<ResCandidate<AbsAppT>, bool> filter)
    {
        if (filter(this))
            return this;
        return null;
    }

    public IEnumerable<ResCandidate<AbsAppT>> AllCandidates
    {
        get { yield return this; }
    }

    public ResCandidate<AbsAppT> BestCandidate { get { return this; } }

    public ResOverloadScore Score { get { return _env.Score; } }

    public bool PostCheckTypeArgs()
    {
        if (_dummyArgs == null)
            return true;

        return _context.CheckDummyArgs(
            _range,
            _dummyArgs,
            _env);
    }
}
```

Figure 6.12: Private IResCandidate interface (top), private abstract ResCandidate base class (bottom) (in ResolveContext.cs)

Figure 6.12 shows the crucial `IResCandidate<AbsAppT>` interface and the abstract base class `ResCandidate<AbsAppT>` of possible derived candidates. `IResCandidate<AbsAppT>` declares a generic interface, which enforces a collection holding all candidates and a C# property to receive the best matching candidate. Further, the `Filter(..)` method allows to specify a lambda expression, which picks a given candidate declaration according to some predicate defined in the filter function. The abstract base class `ResCandidate<AbsAppT>` implements the previously discussed contract and further declares the abstract methods:

- `public abstract bool CheckArity()`
- `public abstract bool CheckTypes()`
- `public abstract IResTerm Gen()`

Implementations of the abstract base class embed the according semantic analysis and resolved term generation tasks in these methods. The `ResolveContext` class supplies two recursive methods to determine the candidates:

- ```
  IResCandidate<AbsAppT> CreateAppCandidates<AbsAppT>(
      Func<SourceRange, IResTerm, ResArg<IResTerm>[],
      ResEnv,
      IResCandidate<AbsAppT>> generator,SourceRange range,
      IResTerm term,ResArg<IResTerm>[] args, ResEnv env)
  ```
- ```
  IResCandidate<AbsApp> CreateValAppCandidates(
              SourceRange range,
              IResTerm term,
              ResArg<IResTerm>[] args,
              ResEnv env)
  ```

For the resolved term the first method recursively looks up the actual term declaration. That is, for `ResMemberCategoryGroupRef`, `ResOverLoadedTerm` and `ResLayeredTerm` instances the according group member-, overload- and layered terms are called, respectively. If the recursion reaches the actual member reference the generator function is called, which is the the second method. The `CreateValAppCandidates(..)` method creates the actual candidate type, which could be an instance of the following classes:

- `ResMethodCandidate`
- `ResAttributeCandidate`
- `ResElemenCtorCandidate`

For the sake of clarification take the following abstract application example used to initialize the variable `a`.

```
@Constant int a = 2 + 3;
```

The right hand side consists of the application of the *plus* operator, which is declared within the `OpenGL42DrawPass` base class in `stdlib.spark`. The following nested scope state is given:

```
ResPairScope
   inner: ResPipelineScope
   outer: ResPairScope
             inner: ResModuleScope
             outer: ResLocalScope
```

The most outer local scope declares the root global scope, where the module scope is nested. The pipeline scope of the `OpenGL42DrawPass` base class is inserted within the module scope. For simplicity, assume that the attribute definition `a` also resides in `OpenGL42DrawPass`. Using the resolve environment the operator *plus* is first looked up in the most inner scope. A `ResLayeredTerm` instance is created, since outer scopes may have matching declarations. Using the `ResLayeredTerm` instance the matching candidates are created with the help of the `CreateAppCandidates<AbsAppT>(..)` method. If the declaration in the most inner layer does not match, outer layers may be browsed for candidates. In this application the function is actually a `ResMethodRef` and can unambiguously be resolved to a `ResMethodCandidate`. Finally, semantic analysis is applied via specifying a filter function, consisting of the `CheckArity()` and `CheckTypes` implementations discussed above. The first method confirms that the exact amount of parameters are passed, and the latter method performs type checking on the parameter types.

Integer And Float Literals

Since the integer and float types are added programmatically before initializing any pipeline declaration, terms denoting integer or float literals are simply represented by `ResLit<T>` instances.

6.7 Resolving Statements

Sequential Statements

A recursive function is implemented to perform semantic analysis on a list of statements (e.g. in a method body). Additionally, a `AbsSeqStmt` class is introduced to compose sequential statements. `AbsSeqStmt` instances consist of a *head*- and *tail* statement.

Figure 6.13 shows the `ResolveStmtRawImpl` method: first the head statement is resolved and a C# lambda expression is used to resolve the tail statement. The artificial `AbsEmptyStmt` class is used to create empty statements, so that once an empty statement is reached in the sequence, the tail statements may be resolved. The simple case of `AbsExpStmt` instances are

```
private IResExp ResolveStmtRawImpl(
    AbsSeqStmt absStmt,
    ResEnv env,
    StmtContext context,
    Func<ResEnv, IResExp> continuation)
{
    return ResolveStmtRaw(
        absStmt.Head,
        env,
        context,
        (e) => ResolveStmtRaw(
            absStmt.Tail,
            e,
            context,
            continuation));
}
```

Figure 6.13: Resolve sequential statements (in ResolveContext.cs)

resolved by resolving the value term and coercing it to an expression. Other types of statements need further explanation, however, most of the statement types invoke the anonymous `continuation` method to initialize the tail statement.

If Statement

An abstract *if-statement* comprises the following properties:

- A term representing the *if-condition*
- A then statement: computed if the condition is fulfilled
- A else statement: computed otherwise

The term indicating the if-condition is usually a function application (c.f. Section 6.6) enforced to return a boolean value. More particularly, the term has to be coerced to the frequency-qualified type of the current resolve context.

For Statement

An abstract *for-statement* comprises a term representing the loop sequence (e.g. `f in Range(0,10)`) and an abstract statement denoting the actual computation in the body. The loop variable (`f`) is adapted to the current computation rate. For example, a for-statement is used in the *geometry shader* to iterate over the vertices of a primitive. The loop variable is lifted from the unqualified type expression `int` to the frequency qualified type expression `@GeometryOutput int`. Of course, the loop variable has to be nested in the local scope of the for-statement. Finally, the statement of the body is resolved.

Switch Statement

A *switch-statement* consists of a statement value and several possible cases. First the term denoting the statement value is resolved (c.f. Section 6.6) and coerced to an expression. Then the *switch-cases* are resolved. Each case comprises a term representing the case value and an unresolved statement denoting the body, which is evaluated. Again, the case value is resolved and coerced to an expression. Then the body-statement is resolved.

Let Statement

Basically, a *let-statement* consists of two unresolved terms denoting type and value of the let statement. There are two flavors of let-statements. The type is resolved as discussed in Section 6.6 and coerced to a type expression. If the type expression is not a frequency-qualified type and the current context has some assigned computation frequency, the non-frequency-qualified type expression is lifted. That is, the result type expression is enforced to be a frequency-qualified type, otherwise an error message is generated. A resolved variable declaration is created with the according type expression and let-statement name. Then the let-statement value is resolved and coerced to a frequency qualified type. With the resolved variable declaration and the resolved let-statement value a resolved let-expression is initialized.

Return Statement

A return statement is resolved by resolving the value term of the `AbsReturnStmt` instance. Further, the return type is coerced to the current computation rate. The result is an instance of `ResBreakExp`.

CHAPTER 7

Optimization

The performance of the generated shaders mostly rely on the more complete HLSL and GLSL compilers underneath the Spark compiler. However, in [3, Section 4.1.2] Foley argues why some optimizations could still be viable. Here, supplementary to Foley's coarse description, a more complete view on the implementation is provided. In particular, *common sub-expression elimination (CSE)* and *dead code elimination (DCE)* is performed during optimization [3, Section 4.1.2]. Here, we also shed light on the intermediate representation and optimization process in general.

7.1 Build System

The intermediate representation of the optimization phase is generated by deploying a *building facility*, which is immanent to all member declarations via the `Builder` class. Figure 7.1 shows the inheritance relationship between member declarations and the `Builder` class. All member declarations derive from `Builder`. Some member declarations are grouped as containers. For instance, a module declaration holds pipeline declarations, which hold several facet declarations.

IBuilder Interface

Figure 7.2 highlights the `IBuilder` interface declaration. All classes implementing the specified contract provide methods to add *build-* and *post build actions*. Further, interlinked build tasks are maintained by a data structure holding all children, representing subsidiary build tasks. Since `IBuilder` derives from `ILazyFactory`, generic lazy instances can be added and created via

```
ILazy<T> NewLazy<T>(Func<T> generator);
```

The builder becomes available by calling `Force()`. `ForceDeep()` makes the builder and its children available.

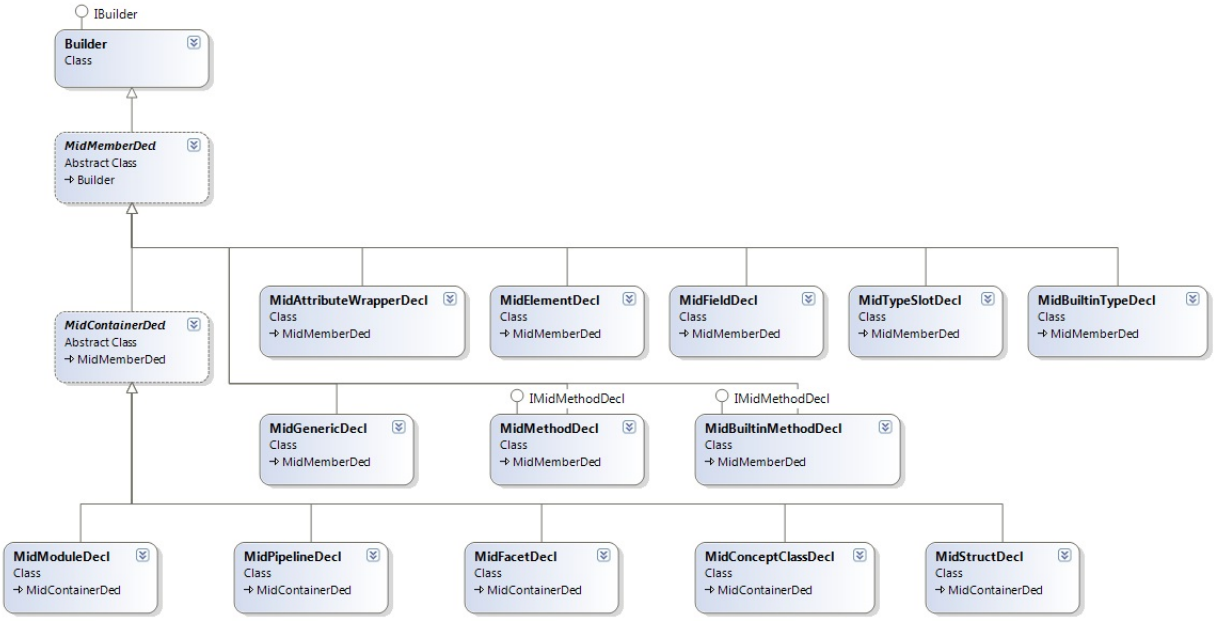

Figure 7.1: Base class Builder and different kind of member declarations

```
public interface IBuilder : ILazyFactory
{
    void DoneBuilding();

    void Force();
    void ForceDeep();
    void AddBuildAction(Action action);
    void AddPostAction(Action action);
    void AddChild(IBuilder builder);

    ILazy<T> NewLazy<T>(Func<T> generator);
}
```

Figure 7.2: The IBuilder interface derives from the ILazyFactory interface known from the semantic analysis module (in Builder.cs)

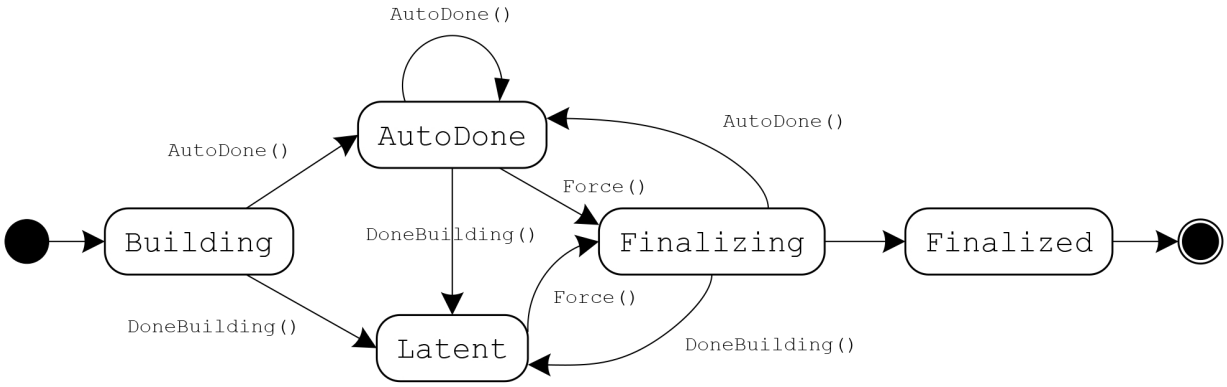

Figure 7.3: Builder State Diagram

Builder States

The Builder has the following states:

- Building:
  - Initial, modifiable state
  - Build- and post actions can be added
  - Children can be added
- Latent:
  - Reached by calling `DoneBuilding()`
  - Only post-build actions can be added
  - Children cannot be added
- AutoDone:
  - Reached by calling `AutoDoneBuilding()`
  - Similar to the latent state, however, builder is still modifiable
- Finalizing
  - Reached by calling `Force()`, which internally triggers `ForceImpl()`
  - Performs build actions
- Finalized
  - Reached after performing build actions
  - Post actions are performed

Figure 7.3 shows the state diagram of the builder. Due to simplification some state transitions are not delineated. Invalid state transitions, for instance calling `Force()` when the builder is already in the finalizing state (circular reference) or adding build actions when the builder is already finalized will trigger an exception. Inappropriate state transitions are handled by the `Force()` and `AssertBuildable()` methods of the builder. However, in the building state actions can be added and children may be appended. When the builder is done, it enters the latent state. That is, the build actions are still not performed. In the finalizing and finalized state the build and post actions are executed.

Emit Environment

The initialization of the new intermediate representation (IR) is attended by derivations of an abstract class `MidEmitEnv`. The following specializations are available:

- `MidGlobalEmitEnv`: often used at module and pipeline scope
- `MidDummyEmitEnv`: intermediate container
- `GenericArgEmitEnv`: used for generic declarations
- `MidLocalEmitEnv`: used for member declarations

The base class maintains the following properties:

- An array of inherited declarations
- A map of resolved- and emitted label expressions
- A map of resolved variable declarations and reference generator functions
- A map of resolved type parameter declarations and type definitions

There are methods to insert and to perform look-ups in the described data structures. However, two distinct methods `CreateTemp` and `MaybeMoveTemp` are used to unfold expressions. For instance, the `MidLocalEmitEnv` is used to emit attribute and method declarations. The following attribute declaration is decomposed into the following sub expressions.

```
@Constant int foo = 2 + 3 + 5 + 0;
->
@Constant int attr$5 = 2 + 3;
@Constant int attr$6 = attr$5 + 5;
@Constant int attr$7 = attr$6 + 0;
@Constant int    foo = attr$7;
```

Also method applications are decomposed:

```
private MidVal EmitVal(IResExp resExp, MidEmitEnv env)
{
    var midType = EmitTypeExp(resExp.Type, env);
    var midExp = EmitExpRaw(resExp, env );
    return env.MaybeMoveTemp<MidVal>(midExp, midType);
}

private MidPath EmitPath(IResExp resExp, MidEmitEnv env)
{
    var midType = EmitTypeExp(resExp.Type, env);
    var midExp = EmitExpRaw(resExp, env);
    return env.MaybeMoveTemp<MidPath>(midExp, midType);
}
```

Figure 7.4: The emit environment used to decompose expressions

```
public T MaybeMoveTemp<T>(MidExp exp, MidType type)
    where T : MidExp
{
    if (exp is T)
        return (T) exp;

    return (T) (MidExp) CreateTemp(exp, type);
}
```

Figure 7.5: Create temporary variables for the sub-expressions

```
@Constant float someFloat = fooMethodCall(5 + 3);
->
@Constant int      attr$9 = 5 + 3;
@Constant float   attr$10 = fooMethodCall(attr$9);
@Constant float someFloat = attr$10;
```

Figure 7.4 shows two methods of the emit context, which uses the methods of the emit environment to decompose expressions. Basically, children ob `MidPath` cannot be decomposed, so all derivations of `MidExp` who are not sub-types of `MidPath` (c.f. Figure 7.6) are decomposed by creating intermediate variables (c.f. Figure 7.5).

Build Algorithm

The build algorithm starts off with the resolved module declaration and initializes a `MidModuleDecl` (c.f. Figure 7.1). Of course, the resolved pipeline declarations are appended by initializing `MidPipelineDecl` instances. The build action of the `MidPipelineDecl` candidate initializes all facet declarations, and so forth. Calling `Force()` makes the builder available and `ForceDeep()` builds all children. When calling `ForceDeep()` in the `Building` state, first

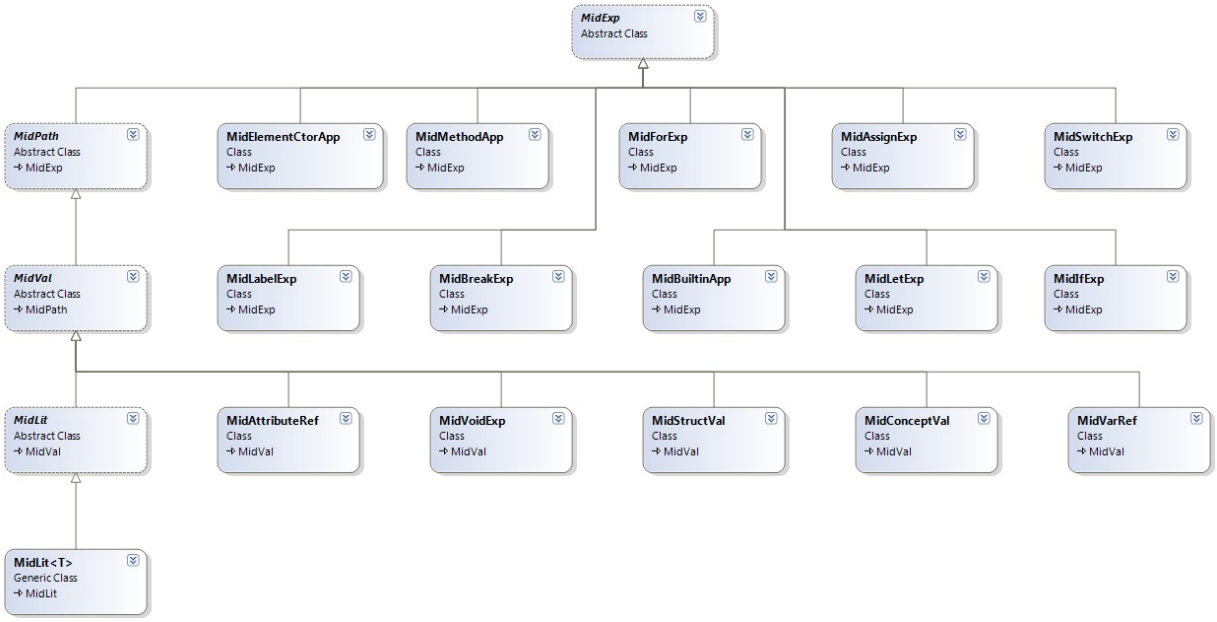

Figure 7.6: All children of MidExp besides children of MidPath are subject to expression decomposition

all parents are initialized, then the private method `ForceImpl()` will perform build and post build actions of the current builder. Finally, all children will be initialized.

7.2 Intermediate Representation

Figure 7.7 underlines the structure of the optimizer. First, with the help of the `MidEmitContext` class, which implements the converting from resolved declaration to the next intermediate representation, the semantically checked module declaration is prepared for the optimizer.

Module Declaration

The resolved module declaration contains the global pipeline declarations. By deriving from `MidContainerDecl` each `MidModuleDecl` holds also all member declarations of the pipeline classes. Neither build- nor post actions are attached to the module declaration.

Pipeline Declaration

For a given resolved pipeline declaration (`IResPipelineDecl`) the `MidPipelineDecl` object is created. The `MidEmitContext` class has the private method

```
MidMemberDecl EmitMemberDeclImpl(MidModuleDecl midModule,
    IResPipelineDecl resPipeline, MidEmitEnv outerEnv)
```

Figure 7.7: Generation of a new IR, CSE and DCE (in MidEmitContext.cs)

which associates the new pipeline declaration with the module declaration used by the optimizer. The `MidPipelineDecl` class provides the following direct attributes

- `DirectFacet`: the direct facet declaration
- `Elements`: the element declaration of the current pipeline declaration (e.g. Constant, Uniform, etc)
- `Facets`: direct and inherited facet declarations
- `IsAbstract`: concreteness mode of the shader class
- `IsPrimary`: shader class type (mixin or primary)
- `Methods`: all method declaration in the class

Beside these essential attributes, of course, each pipeline declaration is associated with a source range and name. Additionally, the class declaration provides the following direct public methods:

- `MidFacetDecl AddFacet(MidPipelineRef originalPipeline)`: adds the facet of another pipeline declaration

```
midPipeline.AddBuildAction(() =>
{
    foreach (var resFacet in resPipeline.Facets)
    {
        var originalPipeline = (MidPipelineRef)EmitMemberTerm(
            resFacet.OriginalPipeline.MemberTerm,
            env);
        var midFacet = midPipeline.AddFacet(
            originalPipeline);

        foreach (var resLine in resFacet.MemberLines)
        {
            var resDecl = resLine.EffectiveDecl;
            var midDecl = EmitMemberDecl(midFacet, resDecl, env);
            midPipeline.InsertMemberDecl(resDecl, midDecl);
        }
        midFacet.DoneBuilding();
    }
});
midPipeline.DoneBuilding();
```

Figure 7.8: C# lambda expression specifying the build action of a pipeline declaration (in MidEmitContext.cs)

- `IMidMemberRef CreateRef(MidMemberTerm memberTerm)`: creates reference to the pipeline declaration for a given member term

For the sake of simplicity the derived attributes and methods are mentioned and described as required.

Figure 7.8 shows the lambda expression which specifies the build action for the pipeline declaration. For each resolved facet declaration of the resolved pipeline declaration new data structures are initialized. Further, each member declaration of a resolved facet declaration (e.g. attribute- or method declarations etc) is emitted by a corresponding `EmitMemberDecl(..)` implementation. The `MidPipelineDecl` class holds both the facets and the member declarations. After adding the build action, the builder transitions to the latent state by calling `DoneBuilding()`.

Facet Declaration

So far the `MidModuleDecl` class holds all new pipeline declaration classes with corresponding build actions. Figure 7.7 shows that after emitting all pipeline declarations the module builder transitions to the latent state and as the next step `ForceDeep()` is called, which makes the module and all children available. That is, the build actions of the pipeline declaration are triggered. Figure 7.8 shows that the build action of a pipeline declaration adds all facet declarations. That is, the pipeline declaration builder, which is in the `Finalizing` state, is extended

by a new child - the facet builder. The `MidFacetDecl` class has the following direct public attributes:

- `Attributes`: all attribute declarations of a facet
- `Elements`: all element declarations of a facet
- `Methods`: all method declarations of a facet
- `OriginalShaderClass`: a reference to the corresponding shader declaration
- `Pipeline`: the actual shader declaration

The `MidFacetDecl` class exposes direct public methods to add attribute, element and method declarations. Looking at Figure 7.8 consecutively all member line declarations are initialized. In the next subsections the implementation of all types of member line declarations is discussed.

Type Slot Declaration

For a type slot declaration there are two different implementations. Either the type slot declaration is preceded by built-in tags and a `MidBuiltinTypeDecl` is initialized, otherwise a `MidTypeSlotDecl` class is initialized. Further, built-in type declarations may expose type parameterization.

The `MidBuiltinTypeDecl` class simply is a wrapper of a `MidBuiltinType`. The `MidBuiltinType` exposes the following direct public attributes:

- `Args`: arguments of the type declaration (generic parameters, size definition etc.)
- `Tags`: the built-in tags of a type declaration

A generic type declaration like:

```
[[Builtin("hlsl", "__Array")]]
[[Builtin("c++", "__Array")]]
[[Builtin("llvm", "__Array")]]
type Array[type T, @Constant int Length];
```

is considered as a generic declaration, with an inner type slot declaration.

Generic Declaration

The `MidGenericDecl` class exposes the following direct public attributes:

- `InnerDecl`: the resolved inner declaration
- `ResDecl`: the resolved generic declaration

- `Env`: the emit environment
- `Context`: the emit context

An example for a generic declaration is given in Section 7.2.

Attribute Wrapper Declaration

Each resolved attribute declaration initializes a attribute wrapper declaration in the optimization phase. The `MidAttributeWrapperDecl` class exposes the following direct public attributes:

- `Attribute`: the wrapped attribute declaration
- `Type`: the type of the attribute declaration

Additionally, the wrapper class exposes a direct public method to create references for according member terms. The wrapper builder is linked to a complex build action, which initializes the actual attribute declaration. Two major cases are distinguished: Either an initializer is provided for the definition or the attribute declaration is uninitialized.

Attribute Declaration

The build action of an attribute wrapper declaration initializes the actual attribute declaration. Since most calculations can be described using attributes in Spark, the initialization requires handling of complex expressions. While non-initialized attribute declarations involve only the preparation of the frequency qualified type of the attribute declaration (c.f. Section 7.5), attribute definitions with initialization need to unfold the right hand sides.

For the simplified case of an initialized attribute definition

```
@Constant int a = 2 + 3;
```

the left hand side exposes a frequency qualified type declaration (`@Constant int`) and the right hand side uses the built-in operator

```
[[Builtin("glsl", "({0}) + ({1})")]]
int operator+( int left, int right );
```

to initialize the attribute declaration. The element declaration `Constant` must already be registered as a member declaration in the pipeline declaration containing the attribute declaration. Next the type expression `int`, which is actually a reference to the type slot declaration `int`, has to be emitted as described in Subsection 7.5. For uninitialized attributes the `MidAttributeDecl` class is initialized using the data structures emitted by the resolved frequency qualified type expression.

Method Declaration

There are two types of method declarations. Either only a method signature is provided with additional built-in tags corresponding to profiles and templates in the underlying architecture (d3d11, opengl4.2, c++, llvm,...) or a usual method definition with probably a method body. Therefore the according `MidBuiltinMethodDecl` or `MidMethodDecl` classes are initialized. The `MidMethodDecl` class exposes the following direct public attributes:

- `Body`: the expression specifying the method body
- `Parameters`: the method parameters
- `ResultTpye`: the result type of the method

The `MidBuiltinMethodDecl` additionally manages built-in tags. Built-in method declarations are not added to the methods of the current facet declaration, but are registered as member declaration in the superior pipeline container.

The optimization phase is centered around attribute- and method definitions. Beside emitting the return type expression and initializing the method parameters, most prominently the method body is optimized.

Element Declaration

The `MidElementDecl` class has the following direct public attributes:

- `Attributes`: all attributes with the corresponding computation rate
- `AttributeWrappers`: all attribute wrappers
- `Outputs`: all output attributes with the corresponding computation rate

Beside direct public methods to add attribute and attribute wrappers, the `MidElementDecl` class provides methods to create attributes for a given expression and type declaration.

The builder of a element declaration has neither build nor post actions to perform. It is simply added as a child to the superior facet declaration.

Struct and Field Declarations

```
struct PNU
{
  vec3 position;
  vec3 normal;
  vec2 texCoord;
}
```

For each resolved struct declaration a `MidStructDecl` instance is initialized, which exposes

```
MidEmitEnv env = new MidGlobalEmitEnv(outerEnv, outerEnv.Context);
env.Insert(
    resPipeline.ThisParameter,
    (SourceRange r) =>
    new MidLit<object>(r, null, new MidPipelineRef(midPipeline, null)));
midPipeline.Env = env;
```

Figure 7.9: This Parameter of the pipeline class is inserted into the global emit environment

- An array of `MidFieldDecl` instances
- and an Identifier denoting the struct name

The build action, which is attached to a `MidStructDecl` iterates over the struct members and performs the according emit implementation for all different kinds of member declarations. All field declarations are uninitialized, therefore, only the type expression is emitted.

Concept Class Declaration

```
concept Linear[type T]
{
abstract T operator+( T left, T right );
abstract T operator-( T left, T right );
abstract T operator*( T left, float right );
}
```

A concept class declaration is a container that defines a contract by specifying certain member declaration signatures, which have to be supported. In the example above the generic *Linear* concept declares three abstract method declarations, addition, subtraction and multiplication, which have to be supported. During the optimization phase a `ResConceptClass` instance is adapted to a `MidConceptClass` declaration. There are neither build actions nor optimization tasks attached to this transition.

7.3 Emitting Expressions

Variable Reference

For each resolved variable reference the emit environment maintains a generator function to initialize the according type of variable reference.

Figure 7.9 shows that for the *this* parameter of the shader class a lambda expression is specified, which generates an instance of `MidLit<object>` for a specific variable reference in the source code.

Attribute Reference

A resolved attribute reference is initialized by emitting the member term property (c.f. Section 7.4). Then the attribute type expression is emitted (c.f. Section 7.5) to, finally, initialize the `MidAttributeRef` instance.

Resolved Literal

```
virtual output @Fragment bool PS_CullFragment = false;
```

The right hand side of the above declaration in the standard library is resolved to an instance of `ResLit<bool>: false` in the abstract syntax tree. After emitting the type expression (c.f. Section 7.5), these nodes are added directly to the expression factory.

Method Application

Each resolved method application is tested whether the result type or the parameters are frequency qualified types. For the simple case of non-frequency qualified types, the resolved member term of the method application is emitted (c.f. Section 7.4). Subsequently, the values of the arguments are initialized. Finally, a reference to the method declaration is initialized and returned. For methods with frequency qualified parameter types or return value the method body is initialized.

Label Expression

For a resolved label expression the label type is emitted (c.f. Section 7.5). The label body is inserted into a local emit environment and the label body is emitted.

Break Expression

First the label of the break expression is looked up in the current emit environment. Then the expression value is emitted to initialize a `MidBreakExp` instance.

Field Reference

```
struct PNU
{
    vec3 pos;
    vec3 normal;
    vec2 texCoord;
}
...
@AssembledVertex PNU vertex;
@AssembledVertex vec3 m_pos = vertex.pos;
...
```

For the above structure declaration, a variable `vertex` is declared at `AssembledVertex` rate. To fetch the position field from the structure the fetch object from the field reference is emitted (`vertex`). Then it has to be ensured that the member term of the container is emitted. Finally, the field reference is added to the expression factory to be used in the simplification stage.

Element Constructor

An element construction expression is similar to a method application:

```
input @RasterVertex int vertexID;
input @RasterVertex int faceID;
...
@FineVertex RasterVertex rVert =
    RasterVertex(vertexID : i, faceID : gl_InvocationID);
...
```

A `ResElementCtorApp` exposes constructor arguments and a computation rate (element). A `MidElementCtorApp` instance is created by emitting the element type expression (c.f. Section 7.5) and the constructor arguments, which are initialized by emitting the member term of the (input) attribute declaration and the initialization value.

Assignment

```
// ...
d = d * ramp;
//...
```

An assignment expression exposes an assignment destination and a source, which is a method application in this case. To initialize an instance of `MidAssignExp` destination and source values are emitted.

Attribute Fetch

The following fragment of the spark standard library shows a plumbing operator, where attributes are converted from `@AssembledVertex-` to `@CoarseVertex` rate.

```
// in stdlib.spark
// ...
input @CoarseVertex AssembledVertex _ia2vs;
// ...
implicit @CoarseVertex T IA2VS[type T]( @AssembledVertex T value )
{
    return value @ _ia2vs ;
}
// ...
```

Some value is projected out of the `_ia2vs` attribute as discussed in [2, Section 2.6], [3, Section 3.3.6] and [3, Section 5.1.6].

```
@AssembledVertex vec3 pos;
@CoarseVertex vec3 m_pos = IA2VS[vec3](pos); // return pos @ _ia2vs;
```

For the above setting, where the position of a vertex is given by the `vec3 pos` attribute, the compiler implicitly inserts the `IA2VS[vec3]` plumbing call, which projects `pos` out of the `_ia2vs` attribute. In the process of initializing the `MidAttributeFetch` instance, the fetch object is emitted.

Concept Class

An instance of a concept class (e.g. `Linear[vec3]`) is initialized by emitting the corresponding type expression (c.f. Section 7.5) and the member term properties of the concept class.

For Expression

The following method specifies the crucial method of an instanced geometry shader.

```
override @GeometryOutput void GeometryShader()
{
    for(i in Range(0,3))
    {
        EmitVertex(RasterVertex(vertexID : i, faceID : gl_InvocationID));
    }
    EndPrimitive();
}
```

An instance of `MidForExp` is initialized for the for expression in the method body. First the type expression of the sequence variable is emitted (c.f. Section 7.5). Finally, the sequence- and loop body expressions are emitted.

Sequential Expressions

A `ResSeqExp` holds a head and a tail expression. There is no associated new data structure for sequential expressions. Simply, the head- and tail expressions are emitted and latter is returned.

7.4 Member Term

Member Bind

A resolved member bind class attaches a resolved member declaration to the *this* pointer of a shader class. For instance, the type declaration `type uint;` in the standard library is bound to the `this#0` variable, which references the `OpenGL42DrawPass` class. The member term is emitted in three steps:

- Emit the value of the bind object
- Emit the member term of the resolved member declaration
- Return a reference to the emitted member declaration (a lookup in the according container is performed)

Global Member Term

A global member term (`ResGlobalMemberTerm`) is emitted, if the current module declaration does not expose the according `MidMemberDecl` property. The module container directly maps resolved member declarations to `MidMemberDecl` instances. After emitting the member declaration, all corresponding build tasks and dependencies are carried out.

Generic Application

A generic method application is initialized by emitting the global member term (c.f. Section 7.4) of the resolved member declaration. Additionally, the generic type parameters are emitted.

Values of expressions

For a given resolved expression the value is emitted by emitting the type expression (c.f. Section 7.5) of the actual expression and then emitting the expression itself (c.f. Section 7.3).

7.5 Type Expressions

Type Slot Reference

When initializing a type slot reference the current pipeline declaration must contain the corresponding member declaration. If this is not the case a referenced type slot member declaration is emitted as described in Subsection 7.2. However, the `ResMemberBind` property is emitted as discussed in Section 7.4.

Pipeline Reference

For a given pipeline reference the type expression is simply emitted by emitting the global member term of the pipeline reference (c.f. Section7.4).

Frequency Qualified Type Expression

A resolved frequency qualified type expression is emitted by initializing the type slot reference indicated by the type expression of the frequency qualified type (c.f. Section 7.5).

Element Reference

```
concrete element AssembledVertex;
... // define @AssembledVertex attributes
input @CoarseVertex AssembledVertex _ia2vs;
```

The unusual but characteristic attribute declaration from above, declares an attribute of type `AssembledVertex` with the `@CoarseVertex` computation rate. As pointed out in [2, Section 2.5] element declarations are similar to C++ struct declarations. Therefore, `_ia2vs` makes all computations at `AssembledVertex` structure available at a `CoarseVertex` rate. Essentially, the new intermediate representation is generated by emitting the `ResMemberBind` term of the resolved element reference (c.f. Section 7.4).

Type Variable Reference

```
// ... built in tags
implicit @Uniform T C2U[type T](@Constant T value);
```

The above method signature declares an implicit generic method, which is inserted by the compiler whenever computations are converted from `@Constant-` to `@Uniform` rate. The return type is parameterized by a certain type `T`, which is given by a `ResTypeVarRef` instance. The following public properties are exposed by this class:

- `Classifier`: The kind classifier of the type
- `Decl`: The actual resolved type parameter declaration
- `Kind`: Type classifier
- `Range`: Source range of occurrence

The emit environment maintains a mapping from type parameter declarations to the actual types. For instance, a fictional `T#10` could be mapped to a `mat4` type.

Void Type

```
abstract mixin shader class OpenGL42GeometryShader
    extends OpenGL42DrawPass
{
  //...
  abstract @GeometryOutput void GeometryShader();
  //...
}
```

The essential functionality of the geometry shader is defined by the implementations of the abstract `GeometryShader` method, which returns a void type. For each `ResBottomType` instance a `MidVoidType` is initialized.

Dummy Type Argument

Instances of `ResDummyTypeArg` are used whenever generic parameter types are specified. For instance, consider the following situation (HLSL example used here):

```
[[Builtin("hlsl", "SamplerState")]]
[[Builtin("c++", "ID3D11SamplerState*")]]
// ... other tags
type SamplerState;
```

The standard library defines a type `SamplerState`, which is mapped to `"ID3D11SamplerState*"` in Directx 11. It encapsulates the state of the sampler for a texture. Further, a method signature is defined to create a `SamplerState` instance:

```
    [[Builtin("c++", "spark::d3d11::CreateSamplerState(device,
        {0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9})")]]
    // ... other built in tags
    @Constant SamplerState SamplerState(
        @Constant D3D11_FILTER                  filter,
        @Constant D3D11_TEXTURE_ADDRESS_MODE    addressU,
        // ... other settings);
```

Another helper function is defined that takes the most prominent settings and uses default settings for the rest:

```
    @Constant SamplerState SamplerState(
        @Constant D3D11_FILTER filter,
        @Constant D3D11_TEXTURE_ADDRESS_MODE address )
    {
        return SamplerState(
            filter: filter,
            addressU: address,
            addressV: address,
            addressW: address,
            // ... default settings used);
    }
```

Each Spark shader, which uses texture mapping initializes a `SamplerState` instance:

```
@Uniform SamplerState samLinear =
    SamplerState(
      filter: D3D11_FILTER_MIN_MAG_MIP_LINEAR,
      address: D3D11_TEXTURE_ADDRESS_WRAP );
```

However, the `@Constant SamplerState` type has to be converted to a `@Uniform SamplerState`. The compiler inserts an appropriate candidate of the implicit conversion method:

```
// ... built in tags
implicit @Uniform T C2U[type T]( @Constant T value );
```

In this case the above `SamplerState` initialization looks like:

```
@Uniform SamplerState samLinear =
    C2U[SamplerState](
       SamplerState(
          filter: D3D11_FILTER_MIN_MAG_MIP_LINEAR,
          address: D3D11_TEXTURE_ADDRESS_WRAP
          )
    );
```

The generic type parameterization is given by an `ResDummyTypeArg` instance, which essentially holds a reference to a concrete type. Therefore, the type expression of a dummy type argument is initialized by emitting the concrete type reference.

Structure Type

```
struct PNU
{
    float3 position;
    float3 normal;
    float2 texCoord;
}

input @Uniform VertexStream[PNU] vertexStream;
```

The structure `PNU` condenses the vertex attributes: position, normal and texture coordinate of a vertex. The attribute declaration `vertexStream` has the type `VertexStream` of `PNU` instances. That is, the generic type parameter references the structure declaration. In this case, the type declaration is emitted by initializing the resolved member bind property of the structure reference.

Concept Class Reference

```
implicit virtual @FineVertex T CoarseToFineWrapper[type T,
     implicit Linear[T]]
     ( @CoarseVertex T value )
{
    return value( _c2fhelper );
}
```

In the standard library an implicit generic method `CoarseToFineWrapper` is declared to convert coarse vertices to fine vertices. Without going into details, note that the second generic value argument enforces that the `Linear[T]` concept to be provided. The concept class reference is initialized by emitting the member term property.

7.6 Simplification

Module Simplification

With the intermediate representation of the optimization phase and a simplification context all attribute expressions and methods are simplified as indicated in the second part of Figure 7.7.

Figure 7.10 shows the the basic simplification algorithm. For each pipeline declaration of a module all attribute expressions and method bodies are simplified. To simplify an expression the `MidTransform` class is used, which may have an pre- and post transform operation. Basically, for a expression first the pre transform action is applied, then all sub expressions are transformed and finally the post transform is applied. At this stage only a post transform operation is bound to an expression.

7.7 Mark Outputs

As a next step attributes which are used in later stages are marked as output variables. Therefore, each pipeline class of the module declaration and all attributes of any computation rate is reiterated. This time the `MidTransform` class is used with the following pre-transform operation:

Figure 7.11 shows that the attribute expression is traversed until a `MidAttributeFetch` instance is found. In that case the corresponding attribute output flag is set to true. Consider the following situation:

```
@CoarseVertex vec3 attr$166 = _ia2vs.m_pos;
->
@AssembledVertex vec3 m_pos = ...; // set output = true!
```

`attr$166` is obviously an attribute declaration, which projects the position of a vertex `@AssembledVertex` rate, as already discussed earlier.

```
public void SimplifyModule(MidModuleDecl module)
{
    foreach (var p in module.Pipelines)
        SimplifyPipeline(p);
}

public void SimplifyPipeline(MidPipelineDecl pipeline)
{
    foreach (var e in pipeline.Elements)
        SimplifyElement(e);

    foreach (var m in pipeline.Methods)
        SimplifyMethod(m);
}

public void SimplifyElement(MidElementDecl element)
{
    foreach (var a in element.Attributes)
        SimplifyAttribute(a);
}

public void SimplifyAttribute(MidAttributeDecl attribute)
{
    attribute.Exp = SimplifyExp(attribute.Exp, new SimplifyEnv(null));
}

public void SimplifyMethod(MidMethodDecl method)
{
    method.Body = SimplifyExp(method.Body, new SimplifyEnv(null));
}

private MidExp SimplifyExp(MidExp exp, SimplifyEnv env)
{
    if (exp == null)
        return null;

    var transform = new MidTransform(
        null, // no pre-transform
        (e) => SimplifyExpImpl((dynamic)e, env));
    return transform.Transform(exp);
}
```

Figure 7.10: Essential methods of the simplification algorithm (in MidSimplifyContext.cs)

```
MidTransform transform = new MidTransform(
    (e) =>
    {
        if (e is MidAttributeFetch)
            ((MidAttributeFetch)e).Attribute.IsOutput = true;
        return e;
    });
```

Figure 7.11: Mark outputs pre-transform operation (in MidMarkOutputs.cs)

CHAPTER 8

Code Generation

In [3, Section 4.1.3 and 4.1.4] Foley provides only two thin conceptual sections about the HLSL code generation, where he describes how record types are mapped to connector structures and so forth. Therefore, this stage of the compilation process is described in more detail in this chapter, to provide insight into the implementation and to understand how these language concepts are mapped to common procedural shaders. The optimized intermediate representation is translated into the pipeline specific shading language. Also, for each shader, C++ wrappers are generated.

Within this thesis, an OpenGL 4.2 back-end has been integrated into the existing Spark framework. The structure of both the HLSL and GLSL code-generators are similar, therefore, the details provided in this chapter apply to both back-ends. Chapter 9, specifically, summarizes some corner-cases and modifications, which were necessary to implement the OpenGL back-end.

8.1 Emit Context And Target

Similar to the previous phases (`ResolveContext`, `MidEmitContext`), during code generation, the `EmitContext` class provides the implementation. Code generation can be divided into two parts: on the one hand C++ wrappers are emitted, on the other hand actual HLSL or GLSL code is assembled. The `EmitTargetCPP` provides specific tasks needed while targeting C++. A `Span` class is used to assemble the source code. In the next sections further details are provided.

8.2 Module Declaration

Naturally, code generation starts by emitting the module declaration. The module declaration itself does not provide any shading logic, so the `EmitModuleCPP` instance simply allocates a *header* and *source* span and adds the first lines of code to the C++ header and source file (Figure 8.1).

```
_headerSpan.WriteLine("// Automatically generated code. Do not edit.");
_headerSpan.WriteLine( "#pragma once" );
_headerSpan.WriteLine( "#include <d3d11.h>" );
_headerSpan.WriteLine("#include <spark/spark.h>");
_headerSpan.WriteLine("#include <spark/context.h>");

_sourceSpan.WriteLine("// Automatically generated code. Do not edit.");
_sourceSpan.WriteLine("#include \"{0}.h\"", _name);
_sourceSpan.WriteLine("#pragma warning(disable: 4100)");
_sourceSpan.WriteLine();
```

Figure 8.1: Module declaration header and source span (in EmitTargetCPP.cs)

All C++ wrapper classes for the shader declaration of the current module are placed into the same header and source span.

8.3 Pipeline Declaration

The pipeline declaration is the central subject of code generation. Two different classes are maintained for the emitted C++ code:

- Interface class:
  - Declares the interface of the shader
  - Provides getter and setter for uniform shader variables
- Implementation class:
  - Provides methods to initialize, submit (render) and release Spark shader instances

For both classes certain class flags are maintained: *None*, *Internal*, *Hidden*, *Implementation*, *Mixin*. At the beginning the interface class has no flags set, while the implementation class is tagged *Hidden* and *Internal*. While emitting shader classes these flags mark where the generated source will be nested.

- Internal classes: have a separate header and source span (all classes from the standard library)
- Hidden classes: header and source are inserted into the source span of the module
- Implementation classes: the class name is prefixed with `"_Impl_"`
- Otherwise: header and source are inserted into header and source span of the module

```
if (baseClass != null)
    baseClassString = string.Format(" : public {0}", baseClass._name);

_headerSpan.WriteLine("class {0}{1}", _name, baseClassString);
_headerSpan.WriteLine("{");
_headerSpan.WriteLine("public:");
_publicSpan = _headerSpan.IndentSpan();
_publicFieldsSpan = _publicSpan.InsertSpan();
_headerSpan.WriteLine("protected:");
_protectedSpan = _headerSpan.IndentSpan();
_headerSpan.WriteLine("public:");
_privateSpan = _headerSpan.IndentSpan();
_headerSpan.WriteLine("};");

_sourceSpan.WriteLine("// {0}", _name);
```

Figure 8.2: Class declaration source and header span (in EmitTargetCPP.cs)

The initialization of the `EmitClassCPP` class emits the structure of a class declaration in the header span a

While emitting pipeline declarations, the `ShaderClassInfo` class maintains certain properties: the optimized IR, Interface- and Implementation Classes, Direct and Inherited Facets.

Direct Facet

For the interface class the direct facet is emitted. For each attribute a `ShaderAttributeInfo` is initialized, which holds certain information about the attribute. The direct facet consists only of non-abstract attributes. Constant attributes are inserted into the emit environment. For all non-input uniform attributes, which are non initialized, getter and setter methods are created. The attribute type is used to initialize a `EmitTypeCPP` object. Here, the built-in tags in the standard library are used to get the corresponding target type. In this case the built-in tag for the template C++ is queried. Then the `EmitTargetCPP` class provides necessary type information for the template.

```
[[Builtin("c++", "ID3D11DepthStencilView*")]]
// ... other tags
type DepthStencilView;
->
EmitTargetCPP.GetBuiltinType(string template = "ID3D11DepthStencilView*",
    IEmitTerm[] args)
```

The `GetBuiltinType(..)` method returns the proper `EmitTypeCPP` instance, with the according type size and alignment information. From the application point of view the Spark context allocates raw memory, and therefore has to keep track of the exact class size. The

following code fragment would be inserted into the class declaration for a world transformation matrix:

```
class BasicSpark11 : public spark::d3d11::D3D11DrawPass
{
public:
    // ...
    // input @Uniform float4x4 world
    spark::float4x4  GetWorld() { return m_world; }
    void SetWorld( spark::float4x4  value ) { m_world = value; }
    // ...
};
```

Also note that the shader `BasicSpark11` derives from the library class `D3D11DrawPass`. For abstract classes no per-stage shader are emitted.

Mixin Shaders

For mixin shaders the interface class flags are set to *Mixin*. Since mixin shaders are derived classes, also the base facet declarations are inspected. For the primary base class the shader class and interface information is retrieved to initialize the `EmitShaderCPP` object. A pointer to the base class, as well as, the primary facet information is populated for the current shader class information. Generally, the inheritance hierarchy is mapped to C++ in the following pattern:

- For primary base classes direct inheritance is used
- Mixin bases are turned into a *has-a* relationship

```
// In Spark
shader class A              { // ... impl }
mixin shader class B        { // ... impl }
shader class C extends A, B { // ... impl }
->
// In C++
public class C : public A
{
public:
// methods to cast C to A or B
private:
    B* _Mixin_B;
} ;
```

For mixin shaders, however, both primary and mixin bases are translated into a *has-a* relationship.

```
// In stdlib.spark
abstract shader class D3D11DrawPass
                              { // pipeline impl }
abstract mixin shader class D3D11Tessellation
    extends D3D11Drawpass     { // tessellation impl}
abstract mixin shader class D3D11QuadTessellation
    extends D3D11Tessellation { // Quad tessellation impl }
->
// behind the scene C++ translation
class D3D11QuadTessellation
{
public:
    // methods to cast to base classes
    // forwarding methods for base facet
private:
    D3D11DrawPass* _Base_D3D11DrawPass;          // primary base
    D3D11Tessellation* _Mixin_D3D11Tessellation; // mixin base
};
```

Implementation Class

So far the interface class was described and how facets are populated and inheritance hierarchies are mapped to C++. As mentioned the implementation class supports wrappers to initialize, submit and release Spark shaders. So the `CreateClass(..)` method is called with different flags and the base class reference. Typically, the implementation class is marked as *hidden* and *implementation* and the interface class is the base class for the implementation class. For mixin shader implementations the base class is the interface class of the primary base. Header and source span of the implementation class are nested into the source span of the module. Mixin bases are added as private fields in the implementation class. Eventually, the initialize, submit and release methods are created. The initialize method usually performs the following tasks:

- Initialize the non DirectX related private fields (mixin classes)
- Initialize the constant buffer
- Initialize the shaders
  - HLSL shading code is emitted as an array of bytes
  - Compile shader
  - Set input layout
- Initialize the blend state

The release methods simply deallocates the shaders, the constant buffer and the blend state. The submit method performs the following tasks:

- Update constant buffer
- Set depth stencil state
- Set render targets
- Set blend state
- Set the shaders
- Set texture samplers
- Render primitive

8.4 Emitting Shader Code

HLSL code is emitted for each pipeline stage, and there are some considerations necessary for the abstraction used in Spark.

HLSL Emit Context

Similar to other phases, most of the implementation is provided again by an emit context. The `EmitContextHLSL` has central control of all information and intermediate representation, which is necessary to translate to the pipeline specific language.

8.5 HLSL Vertex Shader

While emitting the vertex shader the first task is to determine the computation rate, which is forwarded to the next stage. If neither tessellation nor the geometry stage is active the output will be `@RasterVertex` rate. Further, if tessellation is not active and a geometry shader is provided, the output rate is `@FineVertex`. For the case where tessellation shaders are provided, the vertex shader will emit `@CoarseVertex` rate. Next, for the current output element, all output attributes are collected. With the help of the `EmitContextHLSL` class, which actually manages and fragments the source span of the HLSL code, the connector elements are initialized via `GenerateConnectorType(MidElementDecl element)` in `EmitContextHLSL`.

```
Dictionary<MidElementDecl, IAggTypeHLSL> _elementTypes;
Dictionary<MidElementDecl, ConnectorTypeHLSL> _connectorTypes;
```

The above Dictionaries are maintained by `EmitContextHLSL` and keep track of emitted connector types. Basically, `GenerateConnectorType(..)` creates a `ConnectorTypeHLSL` instance and adds it to the `_elementTypes` and `_connectorTypes`.

```
struct T_FineVertex
{
    float3 a_attr_1942_ : USER_a_attr_1942_;
    float2 a_attr_1961_ : USER_a_attr_1961_;
}
```

The above code fragment shows a possible output connector type, which represents the `@FineVertex` element. Obviously, the fine vertex element holds two user defined output attributes `a_attr_1942_` and `a_attr_1961_`. However, these, at first glance rather cryptic fields, correspond to actual attributes, which are just renamed according the the compiler intern naming convention. So, the first attribute could be the vertex position in world space, which is consumed by the geometry shader. The latter `float2` value could be the texture coordinate. The HLSL vertex shader has the following underlying template:

```
T_OUTPUTELEMENT main(// input @CoarseVertex attributes)
{
 // actual computations
}
```

For the example discussed in Section 3.4 the vertex shader receives assembled vertices from the input assembler and outputs fine vertices to the geometry shader.

```
struct T_FineVertex      { // fields };
struct T_AssembledVertex { // fields };

// vertex shader receives assembled vertices, and the
// two @CoarseVertex input attributes: VS_VertexID and SV_VertexID
T_FineVertex main(T_AssembledVertex __ia2vs, uint VS_VertexID: SV_VertexID,
 uint VS_InstanceID: SV_InstanceID)
{
 //  actual computations
}
```

The `DeclareConnectorAndBind(..)` method is used to generate the connector type and to declare a variable of that type. Further, the declared variable is maintained by the `Dictionary<MidAttributeDecl, EmitValHLSL> _attrVals` dictionary. To perform the computation targeted for the vertex shader and to produce a output stream of `FineVertices`, the connector types have to be initialized and computed. The standard conversion from assembled vertex to coarse vertex is given by the pipeline. Since for the current example the tessellation stage is not active, the vertex shader has to perform the pseudo conversion from coarse to fine vertices using the built-in method in the standard library. In Spark, initializing a record element has the effect of performing all computations at the current rate [3, Section 3.2.3]. The `EmitTempRecordCtor(..)` is used to temporarily create a structure corresponding to the record element and performing all computations locally. The compiler will create a local tuple type, which is added to the `_elementTypes` dictionary.

```
Tup_CoarseVertex { // fields }
```

The computation of the necessary attributes is performed recursively. First all sub-expressions are emitted. For the cube map example in Section 3.4 the `__c2fhelper` attribute is evaluated, since tessellation is not active.

```
[[Builtin("hlsl", "__C2FHelper()")]] @Constant CoarseVertex __C2FHelper();
@FineVertex CoarseVertex __c2fhelper = __C2FHelper();
```

`__c2fhelper` is type of `CoarseVertex`, however, given at `@FineVertex` rate. The initialization is provided by a built-in method `__C2FHelper()`. In the base class of `RenderCubeMapSpark` the vertex position in model space is fetched to transform to world coordinates. Also the texture coordinates are used to sample a diffuse texture to override the abstract `target` attribute of the `Base` class.

```
T_FineVertex main(T_AssembledVertex __ia2vs, ...)
{
    // ... (other computations)
    float2 a_attr_1960_ = (__ia2vs).a_texCoord_;
    float3 a_attr_276_ = (__ia2vs).a_P_model_;

    float a_attr_277_ = ((float) 1);
    float4 a_attr_278_ = float4(a_attr_276_, a_attr_277_);

    float4x4 a_attr_279_ = world;
    float4x3 a_attr_280_ = ((float4x3) a_attr_279_);
    float3 a_P_world_ = mul(a_attr_278_, a_attr_280_);

    float2 __c2fhelpera_attr_1960_ = a_attr_1960_;
    float3 __c2fhelpera_P_world_ = a_P_world_;

    // ... (other computations)
}
```

Note how the necessary attributes are fetched from the `T_AssembledVertex` connector type and computations are performed `@CoarseVertex` rate in the vertex shader. If the geometry shader is also inactive, the `__f2rhelper` attribute in the standard library is initialized to provide `@FineVertex` results. However, with the geometry shader fine vertices are produced in the geometry shader stage.

```
T_FineVertex main(T_AssembledVertex __ia2vs, ...)
{
    // ... (other computations)
```

```
    float3 a_attr_1942_ = __c2fhelpera_P_world_;
    float2 a_attr_1961_ = __c2fhelpera_attr_1960_;

    T_FineVertex __result = {a_attr_1942_, a_attr_1961_};
    return result;
}
```

At the end of the vertex shader the constant buffer is emitted. In HLSL the cbuffer structure holds the uniform variables.

```
cbuffer Uniforms
{
    float4x4 world : packoffset(c0);
}
```

8.6 HLSL Hull Shader

Since Direct3D tessellation shaders may perform group-wise operations, there are a collection of differences when emitting the hull and domain shader (c.f. Section 1.4). First of all, the HLSL hull shader is divided into two parts:

- Constant Hull Shader
  - Evaluated per patch
  - Must output tessellation factors
  - May output other patch data
- Control Point Hull Shader
  - Receives input control points, outputs certain number of control points
  - Evaluated per output control-point
  - Attributes have to be defined: domain, partitioning, output topology, etc

Constant Hull Shader

The compiler first emits the constant hull shaders, which is always labeled as `__patchMain`. The following layout is used when emitting the hull shader:

```
T_OutputPatch __patchMain(InputPatch<T_CoarseVertex, ~> HS_InputCoarseVertices,
 uint HS_PatchID: SV_PrimitiveID)
{
    for( uint HS_CornerID = 0; HS_CornerID < ~; HS_CornerID++ )
    {
       // per corner computations
```

```
    }

    for( uint HS_EdgeID = 0; HS_EdgeID < ~; HS_EdgeID++ )
    {
       // per edge computations
    }
    // compute tessellation factors
    // and supplementary patch data (e.g. pn triangles)
}

[domain(~)]
[partitioning(~)]
[outputtopology(~)]
[outputcontrolpoints(~)]
[patchconstantfunc("__patchMain")]
T_ControlPoint main(
 InputPatch<T_CoarseVertex, ~> HS_InputCoarseVertices,
 uint HS_PatchID: SV_PrimitiveID,
 uint HS_ControlPointID: SV_OutputControlPointID)
{
    // e.g. input triangle patch with 3 control-points
    //      output cubic Bezier triangle patch with 10 control-points
}
```

Note that from now on ~ denotes a variable, which may be different from shader to shader. The standard library defines the auxiliary attribute `__ip2op` and provides the constant hull shader with a fixed size array of type `InputCoarseVertexArray` of coarse vertices. The generic `InputCoarseVertexArray` type corresponds to the generic `InputPatch` type in HLSL. The compiler emits the `__ip2op` attribute and triggers the invocation of the `InputPatch` element in the method body of `__mainPatch(..)`. That is, all `@InputPatch` computations are performed.

```
// in stdlib.spark
abstract mixin shader class D3D11Tessellation
    extends D3D11DrawPass
{
    // ...
    input @OutputPatch InputPatch __ip2op;
    abstract output @Constant int HS_InputCoarseVertexCount;
    input @__InputPatch
    InputCoarseVertexArray[CoarseVertex, HS_InputCoarseVertexCount]
        HS_InputCoarseVertices;
    // ...
}
```

Next, a for-loop is emitted to embed per corner computations (`@PatchCorner`). The upper bound of the loop is determined by the `HS_PatchCornerCount` attribute. The derived classes `D3D11TriTessellation` and `D3D11QuadTessellation` provide according settings. The `HS_PatchCornerID` attribute is actually bound to the loop variable `HS_CornerID`. The `HS_PatchCorners` attribute is fetched out of the `OutputPatch` connector type and the `@PatchCorner` computations are emitted.

Analogically, a for-loop is emitted for per edge computations. The upper bound is given by the `HS_PatchEdgeCount` attribute. The `HS_PatchEdgeID` is bound to the loop variable `HS_EdgeID`. The `@PatchEdge` computations for the local `HS_OutputPatch` are computed, that is, at least edge factors are determined. For the inner tessellation factors and `@PatchInterior` computations an additional loop is emitted depending on the primitive type, which is tessellated. Finally, all `@OutputPatch` computations are emitted and the results are stored in the `HS_OutputPatch` variable.

Control Point Hull Shader

For the control point hull shader, of course, the tessellation attribute (e.g. primitive type, number of output control points, ...) are emitted. As mentioned, the control point hull shader receives a certain amount of control points and produces a defined number of control points. The input control points are given by `InputPatch<T_CoarseVertex, ~> HS_InputCoarseVertices`. Additionally, `HS_ControlPointID` refers to the output control point of the current invocation and `HS_PatchID` indicates the current patch of the draw span. Inside the method body the `@InputPatch` computations are emitted locally, and the control point is initialized (`@ControlPoint`)

8.7 HLSL Domain Shader

```
[domain(~)]
T_~ main(
 T_OutputPatch __op2dv,
 float3 DS_DomainLocation: SV_DomainLocation,
 OutputPatch<T_ControlPoint, 3> DS_InputControlPoints )
{
    // interpolate vertex attributes
    // project fine vertices etc
}
```

If tessellation is active and no geometry shaders are used, the domain shader returns raster vertices, otherwise fine vertices are computed. Again, the domain shader is invoked for each parametric location created by the tessellation unit according to the tessellation parameters. Additionally, it receives an array of control points and also the output patch data from the constant hull shader. Typically, the actual fine vertex positions have to be interpolated given the input data and parametric locations. Initially, the `ControlPoint` and, depending on the output

element type, either `FineVertex` or `RasterVertex` connector types are emitted. If no geometry shader is specified, the auxiliary `__f2rhelper` attribute is emitted to plumb fine vertex attributes to raster vertex attributes to, eventually, perform `RasterVertex` computations. Otherwise, simply the output element type is emitted.

8.8 HLSL Geometry Shader

Spark shaders addressing the geometry shader stage derive from `D3D11GeometryShader` and expose an attribute `__D3D11GeometryShaderEnabled`. Since the tasks of the geometry shader differs from the traditional vertex and fragment shaders the shader template is quite different:

```
[instance(~GS_InstanceCount~)]
[maxvertexCount(~GS_MaxOutputVertexCount~})]
void main(triangle T_FineVertex GS_InputVertices[~GS_InputVertexCount~],
          inout TriangleStream<T_RasterVertex> GS_OutputStream,
          uint GS_InstanceID : SV_GSInstanceID)
{
    // Geometry shader method implementation
}
```

Each spark shader targeting the geometry shader stage exposes an attribute corresponding to the geometry shader instance count, the number of in- and output vertices. Currently, only triangles or triangle strips are accepted.

```
// in stdlib.spark
abstract shader class D3D11DrawPass
{
    // ...
    [[Builtin("hlsl", "TriangleStream<{0}>")]]
    type Stream[type T];
    // ...
}

abstract mixin shader class D3D11GeometryShader
    extends D3D11DrawPass
{
    // ...
    input @GeometryInput
        Array[FineVertex, GS_InputVertexCount] GS_InputVertices;
    // ...
    input @GeometryOutput Stream[RasterVertex] GS_OutputStream;
    // ...
}
```

The input is defined as an array of fine vertices in the standard library, whereas the output corresponds to the built-in `Stream` type, which is mapped to the generic `TriangelStream` type in HLSL. The `inout` prefix has to be specified since the output stream is written directly. In the geometry shader body, first, the `__gi2go` attribute is emitted by invoking the `GeometryInput` element constructor. For the example in Section 1.4 no output attributes are given, so nothing is emitted. Next, the `GeometryOutput` element constructor is invoked. In the standard library the following declaration is given:

```
// in stdlib.spark
abstract mixin shader class D3D11GeometryShader
    extends D3D11DrawPass
{
    // ...
    abstract @GeometryOutput void GeometryShader();
    output @GeometryOutput void __GeometryOutput = GeometryShader();
    // ...
}
```

The functionality of the geometry shader is modified by overriding the abstract `GeometryShader()` method. The `__GeometryOutput` output attribute triggers that the custom geometry shader method is emitted into the HLSL geometry shader body.

```
// in CubeMapGS.spark
shader class RenderCubeMapSpark extends RenderToCubeMapBase
{
    // ...
    override @GeometryOutput void GeometryShader()
    {
        for( f in Range(0, 6) )
        {
            for( v in Range(0,3) )
            {
                Append( GS_OutputStream,
                    RasterVertex( fineVertex: GS_InputVertices(v),
                                  cubeMapFaceIndex: uint(f) ) );
            }
            RestartStrip( GS_OutputStream );
        }
    }
    // ...
}
```

The above implementation of a cube map algorithm uses two for loops to render the incoming triangle according to the six faces of the cube map. In particular, note that the `RasterVertex`

constructor is invoked 18 times, each with the according vertex and cube map face arguments. The Spark compiler, therefore, emits an additional method according to the element constructor:

```
void Ctor_RasterVertex(
    out  T_RasterVertex __result,
    uint a_cubeMapFaceIndex_,
    T_FineVertex a_fineVertex_)
{
   // @RasterVertex computations
}
```

Finally, the constant buffer is emitted.

8.9 HLSL Pixel Shader

The pixel shader is the last programmable stage in the pipeline, therefore, additional tasks are attached when emitting computations targeting the pixel shader. First of all, the number of render targets have to be extracted out of the `Pixel` element record. Code corresponding to the configurable output merger stage is emitted next. Most prominently, the blend state is initialized for the render targets. Next, the actual pixel shader is emitted.

```
void main(T_RasterVertex __rv2f,
   float4 PS_ScreenSpacePosition: SV_Position,
   out float4 target0 : SV_Target0,
   out float4 target1 : SV_Target1, ...)
{
   // @Fragment computations
}
```

The above code fragment shows the layout of the pixel shader, which receives a `RasterVertex` and the screen space position of the vertex and computes the fragment values for the render targets. The `__ps2om` attribute from the standard library is emitted by locally. A temporary `Tup_Fragment` type is generated, with fields corresponding to `@Fragment` output attributes. At least the values for the render targets are computed.

```
// in stdlib.spark
abstract shader class D3D11DrawPass
{
    // ...
    virtual output @Fragment bool PS_CullFragment = false;
    // ...
}
```

Additionally, the `PS_CullFragment` attribute is emitted. Spark shaders may override the attribute to discard certain fragments.

8.10 HLSL Shader Compilation

At the end of each shader emitting stage the generated code is compiled via .NET framework interoperability. The `EmitShaderSetup(..)` method registers the HLSL compiler wrapper class via C# to C++/CLI transition.

```
[System.Runtime.InteropServices.DllImport("SparkCPP.dll")]
static extern void SparkRegisterHlslCompiler();
```

The `SparkRegisterHlslCompiler` method simply initializes the HLSL compiler implementation and passes it back to the C# context. The generated HLSL shader is passed to the HLSL compiler and the byte code representation is returned. The string representation is emitted into the initialize block of the C++ wrapper class. Further, the shader size and byte code is stored locally to generate the actual shader when invoking the C++ `Initialize(..)` method of the C++ wrapper class. At this time, also, private fields holding the HLSL shader are emitted and clean up code is generated in the `Release()` method.

CHAPTER 9

OpenGL 4.2 Support

As described at the beginning of the thesis, the Spark framework (and language) is considered to be a scientific reference approach to tackle some of the current difficulties in real-time rendering. This work extends the code generation phase with the capability of targeting the OpenGL 4.2 pipeline.

Supplementary to Foley's high-level description of the code-generation process in [3, Section 4.1.2 and Section 4.1.3], Chapter 8 provides detailed information how the C++ wrappers and HLSL shaders are generated for the optimized pipeline- and facet declarations with the help of the pipeline-specific standard library.

This chapter discusses modifications to the standard library that are necessary to map the given Spark types, operators and methods to the OpenGL 4.2 pipeline.

Further, the creation of C++ wrapper classes has to be adopted to provide the initialization, submission and finalization of all GLSL shaders. Most prominently, the generated wrapper classes expose necessary code structures to compile, bind and deallocate a shader. Further, the wrapper code automatically allocates, updates and finalizes the uniform buffer for all necessary uniform variables.

Last but not least, the shader code-generator has to adapt to all corner-cases that arise due to syntactic and semantic differences between HLSL and GLSL. In particular, the mapping of connector types, the usage of uniform blocks and changes to method signatures are discussed in this chapter.

9.1 Compiler Options

New compiler options are introduced to indicate whether to target Direct3D or OpenGL. The `-t` flag is introduced to specify the pipeline target.

```
-t hlsl YourShader.spark    (generate HLSL shaders)
-t glsl YourShader.spark    (generate GLSL shaders)
Shader.spark                (default target is HLSL)
```

Further, the `-d` flag is introduced to specify a debug output path.

```
-d c:\sparkOut -t glsl YourShader.spark
```

The above compiler option indicates to emit GLSL shaders and to flush out all generated code (C++ wrappers and GLSL shaders) to the specified directory.

9.2 Standard Library

The Spark compiler is highly configurable and extendable via the concept of the standard library, which is written in Spark itself. The standard library declares pipeline specific types, methods, conversion operators and so forth. However, initially the root base class is declared to be `D3D11DrawPass`. So, Spark shaders targeting Direct3D will somehow derive from `D3D11DrawPass` and have access to types and methods, and especially presume naming conventions dictated by the Direct3D pipeline and HLSL shading language. For example, the type `float3` corresponds to the HLSL type `float3` and the method

```
float smoothstep( float low, float high, float value)
```

mirrors `smoothstep` in HLSL. So, for Direct3D users, the Spark shading language is much easier to learn, since the Spark shading language literally exposes the HLSL properties via deriving from `D3D11DrawPass`. Now, OpenGL users are accustomed to other types, methods and naming conventions and also there are much more fundamental differences between these APIs. For example, vector and matrix layout (row versus column major), vertex buffers etc. Basically, an important decision has to be made regarding the confusion and conflicts between both APIs. There are different solution possibilities:

- Introduce a new root class, e.g. `DrawPass`
  - `DrawPass` exposes common types, methods etc
  - `D3D11DrawPass` and the root OpenGL class derive from `DrawPass`
  - Derived classes add individual properties
- Declare two separate roots: `D3D11DrawPass` and `OpenGL42DrawPass`
  - Each root class mirrors the according pipeline
  - Spark shaders targeting Direct3D and OpenGL expose different code

The first solution, really, is the ideal way how this situation is solved. However, there are some considerations regarding this approach. First of all, from the application point of view, there is a dependency to either the Direct3D or OpenGL API (c.f. examples in Chapter3). So, the first question is, which convention should be adopted: Direct3D or OpenGL? If the root class adapts Direct3D, OpenGL users will be confused and vice versa. So, this solution just makes sense if the underlying pipelines are completely abstracted, that is, Spark users should

not be aware of Direct3D or OpenGL back-ends and implementation details. Ideally, Spark shaders should be written and behind the scenes, both Direct3D, OpenGL or even other implementation should be target-able. The second solution is, currently, more practical. Direct3D programmers write Spark shaders deriving from `D3D11DrawPass` and OpenGL users will use `OpenGL42DrawPass`. Both groups will find the conventional pipeline properties. This accommodation comes at a price. Spark shaders deriving from `OpenGL42DrawPass` cannot produce HLSL shaders and vice versa. That is, there may be multiple Spark shaders for each possible back-end implementation. In the scope of this thesis the second approach is implemented, since the first ideal solution would require an additional abstraction layer. Currently, as pointed out in Chapter 3, from the application point of view there is a strong dependency to either Direct3D or OpenGL.

9.3 C++ Wrappers

Math Library

Initially, a separate math library is provided for the Spark framework, which provides data structures for vectors and matrices (e.g. float3, float3x3). When targeting OpenGL, behind the scenes the glm[1] math library is used. Beside different include directives, the code base is not polluted with yet another custom math library. Also, glm is well-known in the OpenGL community and works well with GLSL.

Interface To C++

From the application point of view, basically, the initialization is simplified. In the case of Direct3D, the HLSL shaders are created, the input layout is specified and the blend state is initialized. The C++ wrapper class will provide standard getter setter methods for the `input @Uniform` attributes. Also, `@Constant` and `@Uniform` computations are emitted directly in `Submit(..)` method of the shader wrapper. The GLSL back-end performs the according tasks for the OpenGL context, however, the wrapper classes provide additional methods to fetch attribute locations to setup the vertex attribute pointers correctly.

GLSL Shader Compilation

When emitting HLSL shaders, the generated code is compiled by registering the HLSL compiler via managed-unmanaged transitions and the shader byte code is emitted as a local byte array in the `Initialize(..)` method. The OpenGL back-end does not compile the code directly. The source span is emitted as a char array and compilation code is emitted to the source code. Additionally, error recovery code is emitted, so that if there were errors in the emitted GLSL code during the run-time according messages can be displayed.

```
// @AssembledVertex
input @AssembledVertex vec3 pos;
```

[1] http://glm.g-truc.net/

```
input @AssembledVertex vec3 normal;
```

Assume a basic shader has the above vertex attributes, the OpenGL back-end adds the following methods in the wrapper class:

```
GLint getNormalLocation()
{ return glGetAttribLocation(_programHandle, "aanormal_"); }
GLint getPosLocation()
{ return glGetAttribLocation(_programHandle, "aapos_"); }
```

9.4 Standard Library Changes

With the current approach the standard library is extended by several abstract shaders corresponding to the GLSL pipeline stages.

- `OpenGL42DrawPass`: exposes types, methods, conversions
- `OpenGL42GeometryShader`: exposes element records, methods, etc for the geometry shader stage
- `OpenGL42NullTessellation`: defines a standard conversion from coarse to fine vertices. Used when geometry shader is active and tessellation is off.
- `OpenGL42Tessellation`: exposes necessary types, methods for the tessellation shaders
- `OpenGL42QuadTessellation`: specialization for quad tessellation
- `OpenGL42TriTessellation`: specialization for triangle tessellation

Further, all Spark intern variables starting with two underscores (__~) are reduced to single preceding underscore, since OpenGL reserves names containing double underscores.

9.5 Connector Types

Connector type declaration are mapped to *interface blocks* in GLSL. Basically, one output block of a certain stage can be used as an input block in the later stage(c.f. [7, Section 4.3.7].

```
// HLSL
struct T_RasterVertex { // @RasterVertex attributes };
->
// GLSL
out T_RasterVertex { // @RasterVertex attributes
} T_RASTERVERTEX;
```

However, there are some conventions: Vertex shaders cannot declare input blocks and fragment shaders cannot declare output blocks.

```
// input T_AssembledVertex in vertex shader
in vec3 aanormal_;
in vec3 aapos_;
// etc.
```

That is, the members of the above connector type are simply declared as global variables. Similar considerations are necessary for the `@Fragment` computations. The OpenGL back-end automatically treats those exceptions and also adapts the according attribute fetch expressions.

```
// vertex shader method body:
vec3 inputData = (__ia2vs).attr4; // attribute fetch
->
vec3 inputData = attr4; // attr4 is a global variable
```

9.6 Uniform blocks

Uniform variables are grouped into *uniform blocks* in GLSL.

```
cbuffer Uniforms { // @Uniform attributes }
->
// GLSL
uniform VSUniforms
{
  // @Uniform attributes
} vsUniforms;
```

The following naming convention is used to distinguish uniform blocks in different stages. The block-name is prefixed with an abbreviation for the current stage (vs, tcs [tessellation control shader], tes [tessellation evaluation shader], gs, fs): `~Uniforms`. The instance name follows the same pattern, except the prefix is in lower case: e.g. block-name: `TESUniforms`, instance name: `tesUniforms`. To access variable from the uniform blocks the attribute fetch expressions are adapted:

```
// in HLSL
cbuffer Uniforms { float4x4 mvp : packetoffset(c0); }
// in method body
float4x4 aaattr_149_ = mvp

->

// in GLSL
uniform VSUniforms
{
 mat4 mvp;
```

```
} vsUniforms;
// in method body
mat4 aaattr_149_  = (vsUniforms).mvp;
```

9.7 Signature Changes

In HLSL the connector types are first declared as `struct` variables and then inserted into the method:

```
// in HLSL vertex shader
struct T_AssembledVertex { // @AssembledVertex }
struct T_RasterVertex { // @RasterVertex }

T_RasterVertex main(T_AssembledVertex __ia2vs, // ...)
{
  // computations
  T_RasterVertex __result = { // init connector };
  return __result;
}

->

// in GLSL vertex shader
// T_AssembledVertex connector type
// global input variables (@AssembledVertex)

out T_RasterVertex
{
  // @RasterVertex
} T_RASTERVERTEX;

void main(void)
{
 // access global variables
 // perform computations
vec3 aaattr_342_ = (T_ASSEMBLEDVERTEX).someInputData;
// ...
 // set T_RASTERVERTEX connector type e.g.
 T_RASTERVERTEX.aaattr_342_ = aaattr_342_;
// ...
}
```

In GLSL the main method is not allowed to take any parameters. Therefore, instance names are generated for the interface and uniform blocks. Within the `main` method connectors can be

accessed via the instance names.

9.8 Built-in Variables

In HLSL certain variables are extended by a *system-value semantics* syntax.

```
struct T_RasterVertex
{
    float4 aa__RS_Position_: SV_Position;
    // other @RasterVertex attributes
};
```

The `SV_Position` semantic marks the transformed vertex position for the later stage. In GLSL predefined variables are used for these cases.

In the Direct3D part of the standard library the following declaration

```
abstract @RasterVertex vec4 RS_Position;
output @RasterVertex vec4 _RS_Position = RS_Position;
```

triggers the HLSL back-end to mark the variable with the according system-value semantic. That is, pre-defined Spark variables are recognized and extended by the according semantics. The OpenGL part of the standard library exposes the necessary built-in variable of the opengl pipeline. Therefore, the OpenGL back-end directly links pre-defined Spark variables with pre-defined OpenGL variables.

```
out T_RasterVertex
{
   vec4 aa_RS_Position_;  // RS_Position (predef Spark variable)
   // other @RasterVertex attributes
} T_RASTERVERTEX;

// in method body
gl_Position = aa_RS_Position_; // map to predef OpenGL variable
```

CHAPTER 10

Discussion and Conclusion

In conclusion, an evaluation is provided for the new OpenGL 4.2 back-end. The practical significance for the industry is discussed. Particularly, some requirements for a typical game development company are opposed to the potentialities of Spark.

10.1 Evaluation

The OpenGL 4.2 back-end is evaluated by comparing Spark shaders (targeting GLSL) with a similar OpenGL application with custom GLSL shaders. The deviations are reduced to a minimum, so that the actual differences can be measured:

- The performance of the generated GLSL shaders versus the custom GLSL shaders
- The performance of the C++ wrapper classes managing the uniform block, versus the optimized update of uniform variables

The following three techniques were evaluated:

Method	Spark (fps)	GLSL (fps)	Spark (ms)	GLSL (ms)	% Spark vs. GLSL
Simple Shading (19.2k)	109.665	112.158	9.11859	8.91596	+2.27%
Cube Mapping (1.4k)	149.544	159.514	6.68698	6.26903	+6.67%
Quad Tessellation (1.6k)	269.178	279.034	3.71501	3.58392	+3.66%

The simple shading technique unifies three simple Spark modules, which were introduced in Chapter 1 and 3. It combines vertex transformation, per-pixel lighting and a texturing effect. The generated GLSL shaders address the vertex- and pixel shader.

Dynamic cube mapping was discussed in Section 3.4. For the evaluation, instanced geometry shaders were used to render the different faces of the cube map simultaneously. Here, the compiler generates a GLSL geometry shader.

For the quad tessellation evaluation, a distance-adaptive tessellation algorithm was implemented in Spark. The most important notes on the tessellation control- and evaluation shaders may be found in Section 3.5. The compilation of this Spark shader addresses all stages of the OpenGL 4.2 pipeline.

For each shading effect, the number in the parentheses denotes the amount of triangles in the scene. For the tessellation effect, the number denotes the amount of quads (with 16 control-points each) of the initial coarse mesh.

The effects were rendered on a computer with a Geforce GTX 550 TI graphics card and an AMD A8 3.2Ghz CPU. The applications are evaluated in full-screen mode with a resolution of 1280x720 pixels. For the cube mapping algorithm a 1024x1024 cube map texture was used. The results are the mean values of a collection of measurements acquired to render a thousand frames. The comparison indicates that Spark shaders perform within an net average of 4.2 percent of GLSL shaders.

Foley compares the performance of Spark shaders with corresponding HLSL über-shaders with- and without manual dead-code elimination [14].

Method	Model	HLSL, no DCE	HLSL, DCE	Spark	% Spark vs. HLSL
Cube Mapping	Lizard	0.507	0.506	0.507	0%
(1024x1024x6)	Vortigaunt	6.49	4.36	4.37	0%
Render to Screen	Lizard	0.093	0.093	0.091	-2%
(1792x512)	Big Guy	1.12	0.990	1.01	+2%
	Vortigaunt	0.974	0.851	0.867	+2%

The table above summarizes some of his results (measurements in milliseconds) [14]. As indicated, the Spark shaders perform within a 2% range of the optimized HLSL über-shaders. For Spark shaders, global dead-code elimination is performed as part of the compilation phase as described in Chapter 7 [3, Chapter 4.1.2].

The measurements provided in these two tables cannot be compared directly due to obvious differences in the experiment: hardware, rendered models, type of shaders. However, the nominal performance of Spark shaders in comparison to GLSL/HLSL shaders is similar.

10.2 Industry Relevance

Spark is currently available as a research quality shading language. There are still some missing compiler- and Direct3D 11/OpenGL 4.2 features. In [3, Chapter 5] language design and possible future work is discussed. Here, a more practical subject is focused. Are there any benefits to a real-time graphics centered production work-flow and, ultimately, are there some benefits for the industry?

Game Development

The initial research question and aims were proposed to an experienced game developer and the following considerations were expressed. For a game development company it is crucial

that both programmers and visual artists may work on the shaders. For the developer, in some cases, it is important to bypass frameworks, for instance, to create a rapid prototype, or to optimize compiled code. The artists, however, require a well-defined technical abstraction, and mostly work on 3D graphics software. Shading options and parameters have to be available at a graphical user interface level. A potential new shading framework should not change the general work-flow which has been adapted in a team over many years. The framework should be practical and should not introduce unnecessary complexity. For instance, even technical artists should not be required to do C++ programming work to benefit from the novel framework. Also, game development companies provide products for multiple platforms. Usually, for any target platform (PC, XBOX360, PS3, etc) the shaders are optimized.

Now, in case of the Spark framework, the general work-flow remains. Instead of writing HLSL or GLSL shaders, the developers maintain Spark shaders. The application side is similar, some tasks are even automated via C++ wrappers. The Spark framework is easy to bypass and even the compiled HLSL or GLSL sources and blobs are available for optimization. It is conceivable that a production quality Spark compiler could support multiple platforms. That is, the shaders remain the same, the compiler takes over the complex platform optimization.

Paradox C# Game Engine

Recently, a Japanese company (Silicon Studio Corporation[1]) announced a novel C# game engine[2].

The *Paradox* game engine is still under development, however they announced that they provide a shading framework, which was inspired by Spark. The following features are supported for shader composition[3]:

- Extended shading language
- Class mixin and inheritance
- Composition

Similar to Spark, the Paradox engine supports mixins and shader inheritance, by providing a new shading language, which extends the HLSL. Further, data plumbing features are provided, by automatically detecting variables that are used in later stages. Shaders using the extended language may be converted to HLSL or GLSL. There are many other features, most prominently, different platforms are supported:

- Windows desktop and Windows 8 Metro
- Mobile iOS, Android, WP8
- Next-gen Consoles

[1] `http://www.siliconstudio.co.jp/en/`
[2] `http://paradox3d.net/`
[3] `http://paradox3d.net/features/graphics.html#shader`

- Platform independent graphics API

The first beta will be available in Q3 2013.

10.3 Conclusion

The actual assessment whether Spark could be essential and practical for the industry remains open. However, the potential benefits of the Spark framework are obvious. Procedural shading languages were practical and convenient for some years. The increasing complexity and feature sets of the current graphics hardware and the diversity of computer platforms demand a re-evaluation of language design and middle-ware software architectures. Spark was released in 2011. For 2013, the Silicon Studio Corporation announced a game engine which was inspired by Spark. Over the next years, the practicality of new shading frameworks will certainly be decided.

List of Figures

Bibliography

[1] Ravi Sethi Jeffrey D. Ullman Alfred V. Aho, Monica S. Lam. *Compilers: Principles, Techniques and Tools, Second Edition*. 2011.

[2] Tim Foley. Spark User's Guide, 2011.

[3] Timothy John Foley. *Spark: Modular, Composable Shaders for Graphics Hardware*. PhD thesis, Stanford University, 2011.

[4] John Gough. The GPLEX Input Language, Version 1.1.0, 2009.

[5] John Gough. The GPLEX Scanner Generator, Version 1.1.0, 2009.

[6] John Gough. The GPPG Praser Generator, Version 1.3.5, 2009.

[7] Randi Rost John Kessenich, Dave Baldwin. *The OpenGL Shading Language, Language Version 4.2*, 2011.

[8] Randi Rost John Kessenich, Dave Baldwin. *The OpenGL Shading Language, Language Version 4.3*, 2012.

[9] Roland Kuck. Object-Oriented Shader Design. 2007.

[10] Frank D. Luna. *3D Game Programming With DirectX 11*. 2012.

[11] Kurt Akeley Mark Segal. *The OpenGL Graphics System: A Specification (Version 4.2 (Core Profile))*, 2012.

[12] John O'Rorke. *Gpu Gems*, chapter 36, Integrating Shaders into Applications. 2004.

[13] Gerold Wesche Roland Kuck. A Framework for Object-Oriented Shader Design. 2009.

[14] Pat Hanrahan Tim Foley. Spark: Modular, Composable Shaders for Graphics Hardware. 2011.

[15] Naty Hoffman Tomas Akenine-Möller, Eric Haines. *Real-Time Rendering, Third Edition*. 2008.

[16] David Wolff. *OpenGL 4.0 Shading Language Cookbook*. 2011.