Michael Kerber, BSc

Cross-Platform Shader Translation through a Portable Intermediate Representation

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Institute of Computer Graphics and Vision

Dipl.-Ing. Christopher Dissauer
Dipl.-Ing. Wolfgang Moser

Graz, March 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Place, Date Signature

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Ort, Datum Unterschrift

Abstract

The Murl Engine is a native cross-platform application framework designed for platform-independent development of games and multimedia-rich applications. Such frameworks ideally support a vast variety of shading languages to grant programmers as much flexibility as possible. The aim of this work is to design and implement an application that enables the Murl Engine to handle various input and output shading languages. This application utilizes the Vulkan shading language SPIR-V (Standard Portable Intermediate Representation) as an intermediate representation for shader translation. The Murl Engine has a strong focus on performance and provides a tool for offline optimization of GLSL (OpenGL Shading Language) shaders. Therefore, this thesis introduces basic offline optimizations for SPIR-V modules, such as dead code elimination , inline expansion and copy propagation. The performance of these optimized SPIR-V shaders is evaluated with respect to their frame rates and byte sizes.

Keywords: Murl Engine, shader translation, SPIR-V, Vulkan, compiler optimization

Kurzfassung

Die Murl Engine ist ein plattformunabhängiges Framework, speziell entworfen für die Entwicklung von Spielen und multimedia Anwendungen. Solche Frameworks unterstützen, im besten Fall, so viele verschiedene Shader-Sprachen wie möglich, um Entwicklern maximale Flexibilität zu gewähren. Das Ziel dieser Arbeit ist der Entwurf und die Implementierung einer Anwendung, die es der Murl Engine ermöglicht verschiedene Shader-Sprachen, sowohl als Eingangssprachen als auch als Ausgangssprachen zu handhaben. Diese Anwendung verwendet die native Vulkan Shader-Sprache SPIR-V (Standard Portable Intermediate Representation) um eine Sprache, in eine andere, zu übersetzen. Die Murl Engine wurde für höchste Performance entworfen und bietet auch offline Optimierungen für GLSL (OpenGL Shading Language) Shader an. Aus diesem Grund wurden auch im Zuge dieser Arbeit verschiedene Optimierungen für SPIR-V Module implementiert. Zu diesen zählen unter anderem dead code elimination, inline expansion und copy propagation. Die optimierten SPIR-V Shader wurden hinsichtlich ihrer Frameraten und Dateigrößen evaluiert.

Schlagwörter: Murl Engine, Shaderübersetzung, SPIR-V, Vulkan, Compileroptimierung

Acknowledgments

First and foremost I want to thank my supervisors at Spraylight GmbH, Dipl.-Ing. Christopher Dissauer and Dipl.-Ing. Wolfgang Moser who made this thesis possible. Thank you for many helpful advices and your support with excellent knowledge.

Special thanks to Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg for supervising my master's thesis and assisting me during my work.

I would like to express my gratitude to all who helped me completing this work and who accompanied me during my studies.
I feel obliged to thank Benedikt, Richard and Carmen for always having a sympathetic ear.
Further on I want to thank my family for their support and patience throughout the years.

Finally, I must express my very profound gratitude to my girlfriend Isabel for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of writing this thesis.
This accomplishment would not have been possible without you. Thank you!

This master's thesis was developed in cooperation with Spraylight GmbH.

Contents

1 Introduction **1**
- 1.1 Murl Engine 1

2 Related Work **5**
- 2.1 History of Programmable Shading 5
- 2.2 Shader Stages 6
  - 2.2.1 Vertex Shader 7
  - 2.2.2 Tessellation Shader 7
  - 2.2.3 Geometry Shader 8
  - 2.2.4 Fragment Shader 9
  - 2.2.5 Compute Shader 10
- 2.3 Shading Languages 10
  - 2.3.1 OpenGL Shading Language (GLSL) 11
  - 2.3.2 High Level Shading Language (HLSL) 11
  - 2.3.3 Metal Shading Language (MSL) 11
  - 2.3.4 Standard Portable Intermediate Representation (SPIR) 12
- 2.4 Compiler Optimization 18
  - 2.4.1 Dead Code Elimination 18
  - 2.4.2 Inline Expansion 19
  - 2.4.3 Constant Propagation 21
  - 2.4.4 Copy Propagation 22

3 SPIR-V Tools **23**
- 3.1 GLSL Reference Compiler 23
- 3.2 SPIRV-Cross 25
- 3.3 SPIR-V Tools 26
- 3.4 Krafix 26
- 3.5 ShaderC 28
- 3.6 SMOL-V 29

4 Design & Implementation **30**
4.1 SPIR-V Wrapper 30
4.1.1 Command Line Usage 32
4.1.2 Interface Definition 33
4.2 SPIR-V Optimization 35
4.2.1 Dead Code Elimination Pass 36
4.2.2 Inline Functions Pass 40
4.2.3 Copy Propagation Pass 43
4.2.4 Remap Ids Pass 44
4.2.5 Helper Classes 47
4.2.5.1 Basic Block Info Class 47
4.2.5.2 Function Info Class 50
4.2.5.3 Pass Utils Class 50
4.3 Watchpoints 52

5 Results **54**
5.1 Test Environment 54
5.2 Test Cases 55
5.3 Evaluation of Frame Rates 57
5.4 Evaluation of Binary Sizes 59

6 Conclusion **62**

A List of Test Shaders **63**

Bibliography **67**

List of Figures

1.1 The Murl Engine as a cross-plaftorm framework 2
1.2 Structure of the Murl Engine framework 2
1.3 Shader compilation in the Murl Engine 3
1.4 Shader cross-compilation using SPIR-V as an intermediate language . 4

2.1 Graphics pipeline in OpenGL 6
2.2 Displacement mapping using vertex texturing 7
2.3 Tessellation rendering comparison 8
2.4 Shadow volume generation with a geometry shader 9
2.5 Motion blur effect using fragment shaders 10

3.1 Control flow graph created by SPIR-V Tools 28

4.1 SPIR-V Wrapper setup 31
4.2 SPIR-V Tools optimization passes 32
4.3 SPIR-V Wrapper C-interface 35
4.4 Control flow of the dead code elimination pass 38
4.5 Dead code elimination pass class 39
4.6 Control flow of the inline functions pass 41
4.7 Inline functions pass class 42
4.8 Control flow of the copy propagation pass 44
4.9 Copy propagation pass class 45
4.10 Control flow of the remap ids pass 46
4.11 Remap ids pass class 47
4.12 Basic block info class 49
4.13 Function info class 50
4.14 Pass utils class 51

5.1 Output of test shaders Ex01 and Ex19 57
5.2 Performance evaluation part 1 58

5.3 Performance evaluation part 2 58
5.4 Performance evaluation part 3 59
5.5 Binary size evaluation part 1 60
5.6 Binary size evaluation part 2 61
5.7 Binary size evaluation part 3 61

List of Tables

4.1 Supported shader stages 33
4.2 Supported command line arguments 34
4.3 Supported optimization passes 34

5.1 Order of optimization passes applied for evaluation 56

List of Code Examples

1 GLSL fragment shader example 14
2 SPIR-V shader example part 1 15
2 SPIR-V shader example part 2 16
2 SPIR-V shader example part 3 17
3 Dead code example 19
4 Inline expansion example 20
5 Constant propagation example 21
6 Copy propagation example 22
7 Vertex shader converted by glslang 24
8 Vertex shader before conversion with glslang 25
9 Control flow graph example using GLSL 27
10 Control flow graph example using SPIR-V 27
11 GLSL fragment shader before watchpoint injection 52
12 GLSL fragment shader after watchpoint injection 53
13 Vertex shader of the Vulkan Shader Evaluator 56

Chapter 1

Introduction

Until today the mobile market is still growing rapidly. With the increasing processing power of mobile devices, people demand high standards for their multimedia applications. Developers of such applications have to face enormous challenges in terms of development time, resources, maintenance, tools and deployment. As there are various different platforms to target, like iOS, Windows Mobile or Android, developers need to use many different vendor-specific Software Development Kits (SDKs) to create their applications, even the programming languages vary. To work around these problems, the most popular approach is to use a common code base, written in a single programming language and use a cross-platform framework to create native applications for the targeted platforms. During this work, an application was designed and implemented to enhance the Murl Engine, a cross-platform framework to develop games and multimedia-rich applications [43]. This application is meant to improve the Murl Engine in terms of flexibility and support for different shading languages.

1.1 Murl Engine

The Murl Engine is a lightweight native C++ cross-platform multimedia framework, created and maintained by Spraylight GmbH [44]. As already mentioned above it was designed for the development of games and multimedia applications, with a strong focus on flexibility and performance. It allows fast and easy development of applications for Android, iOS, macOS, Windows and Linux as show in figure 1.1.

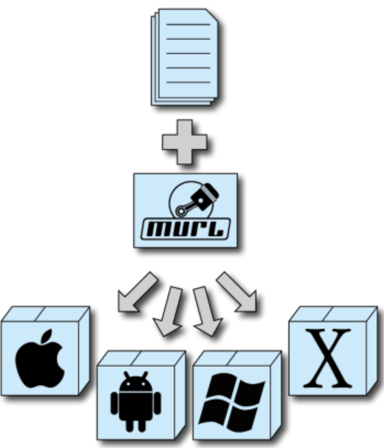


Figure 1.1: The Murl Engine as a cross-plaftorm framework.

Although running on desktop and mobile systems, the Murl Engine is primarily optimized for mobile platforms. Figure 1.2 displays how the Murl framework is structured.

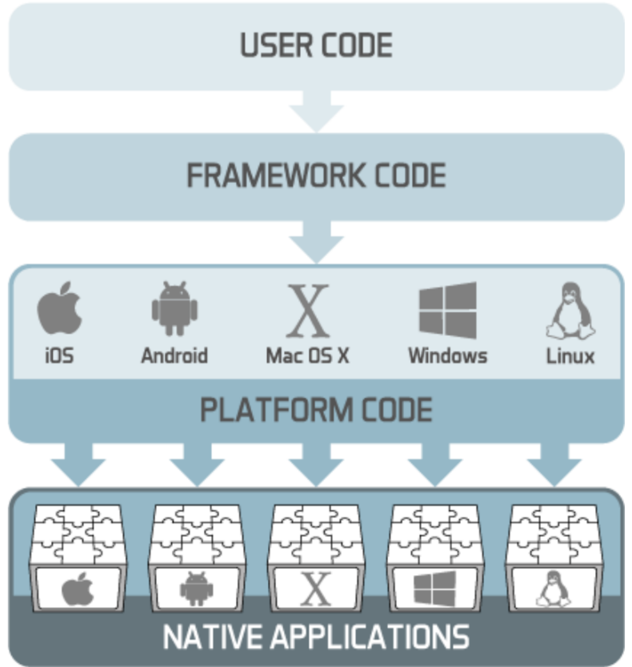


Figure 1.2: Structure of the Murl Engine framework.

As illustrated in Figure 1.2 the code of an application is split into three main parts, the *user code*, the *framework code* and the *platform code*. These three parts combined result in the the native application for the specific platform. While the Murl Engine provides the *framework code* and the *platform code*, the *user code* has to be written by the developer. The *framework code* was developed completely in C/C++, while the *platform code* can either be C/C++, Objective C or Java, depending on the targeted platform. The Murl Engine supports C++ and/or Lua *user code* for application development. As a cross-platform multimedia framework, the Murl Engine also has to support multiple shading languages fitting the targeted platforms and the graphics APIs (Application Programming Interfaces) in use. A windows system using DirectX needs shaders written in HLSL (High Level Shading Language [30]), while a Linux system is dependent on OpenGL and GLSL (OpenGL Shading Language [19]) shaders. Figure 1.3 shows the typical workflow used by the Murl framework, when dealing with shaders. The Murl Engine provides an interface to write GLSL shader code directly within the framework, this user written shader is optimized by an open source tool called GLSL Optimizer, which was created by Aras Pranckevičius [38][39]. The optimized code is then converted to HLSL if needed.

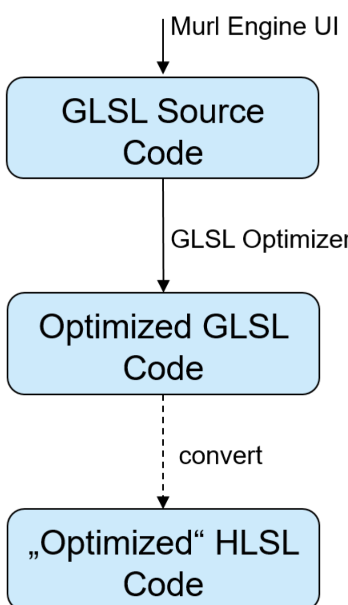


Figure 1.3: Shader compilation in the Murl Engine.

While this works well in terms of performance, it lacks flexibility of supported languages. If a user wants to user another shading language, like HLSL, the Murl

Engine would need to provide and maintain a different optimization tool to achieve optimized shader code. The goal of this thesis is to provide a tool used by the Murl framework to solve this issue. To be independent of the input and output languages, an intermediate language is needed.

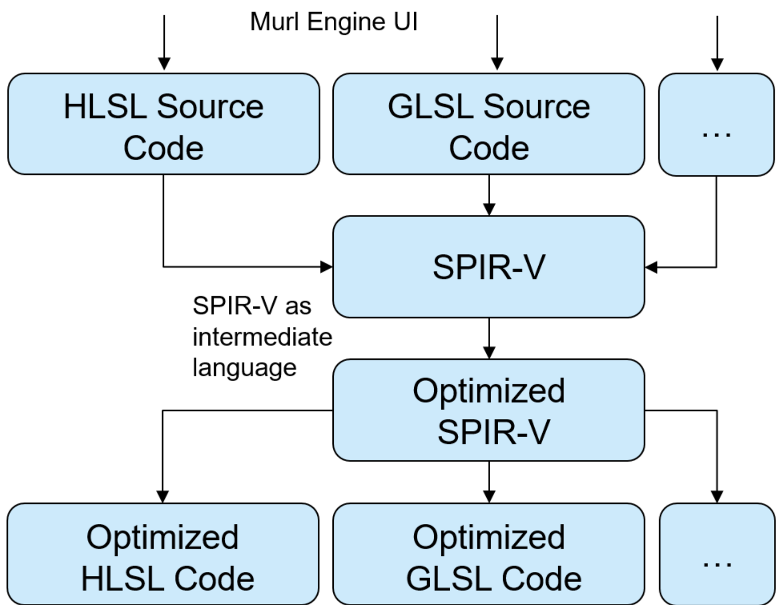


Figure 1.4: Shader cross-compilation using SPIR-V as an intermediate language.

Whatever input language is used, it will be converted to the intermediate representation, then optimized and finally converted back to to output language desired. This is visualized in Figure 1.4, where SPIR-V (Standard Portable Intermediate Representation [21]), which is described in more detail in Section 2.3.4, is used as an intermediate language. Since SPIR-V was designed to be a portable intermediate representation, it fits the needs perfectly. Additionally, as Vulkan is considered the successor of OpenGL and SPIR-V being the native shading language for Vulkan, the Murl Engine will be ready for Vulkan. This approach has the advantage, that while supporting all languages that can be converted to SPIR-V, the optimization is also a lot easier to maintain, as only the SPIR-V code needs to be optimized. The application created during this work, will be part of the Murl Engine as a third-party application.

Chapter 2

Related Work

This chapter covers the most important fundamentals for understanding this thesis. Section 2.1 gives an overview of how programmable shading evolved. Afterwards, section 2.2 deals with the graphics pipeline and its stages, followed by some background to different shading languages in section 2.3. The most relevant compiler optimizations for this thesis are discussed in section 2.4.

2.1 History of Programmable Shading

The need for programmable shading was recognized early on. It was Robert L. Cook who introduced shade trees in 1984 [7]. He saw that shaders based on fixed models are not sufficient for more complex shading tasks and suggested a flexible tree-structured shading model. He further developed his idea and proposed a new rendering architecture in 1987 [8]. This apporach was picked up by Hanrahan and Lawson who introduced one of the first shading languages [13], which later came to be known as the RenderMan Shading Language [1][45]. Even today RenderMan is used for creating visual effects and animations for film productions. Since graphics processing units (GPUs) did not support programmable shading, multi-pass rendering was widley used [34]. The first graphics card to support programmable vertex shaders was the GeForce 3, developed by NVIDIA in 2001 [25]. However, shader programs were very limited in functionality and length back then. They did not allow any branching and had to be written in an assembler-like language. From that point GPUs and also the graphics application program interfaces (APIs), Di-

rectX and OpenGL, improved rapidly. Soon new, higher level, shading languages were needed due to the high complexity of the shaders. With DirectX 9.0 Microsoft and NVIDIA introduced their High Level Shading Language (HLSL [30]) and the cross-platform language Central Graphics (Cg [28]). In parallel, the OpenGL Architecture Review Board (OpenGL ARB) also developed a c-like shading language for their API, called OpenGL Shading Language (GLSL [40]). At this point these high level languages allowed for programming vertex shaders and fragment shaders (or pixel shaders) only. Later on in 2007 shader programming evolved further with the introduction of geometry shaders and Shader Model 4.0 [5]. As a result the fixed-function graphics pipeline was now only used to support older graphics cards and the the programmable shader pipeline became standard. With Direct3D 11 and OpenGL 4.0 tessellation shaders and compute shaders were added to the rendering pipeline [46][48] which leads to the modern programmable graphics pipeline shown in Figure 2.1.

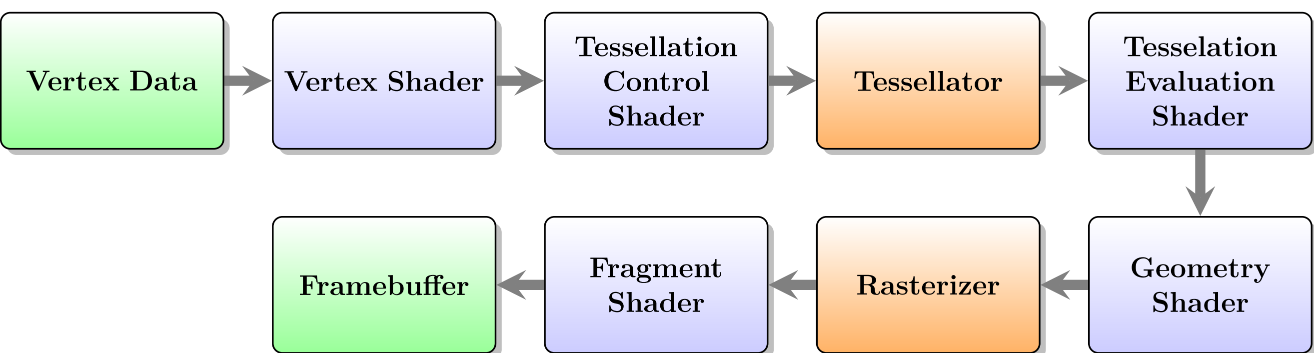


Figure 2.1: The programmable graphics pipeline, as used in OpenGL. The green blocks represent the input and output data. The blue blocks show the programmable stages, while the orange blocks are fixed stages.

2.2 Shader Stages

This section describes the shader stages in modern graphics pipelines. Each shader type is explained in more detail, along with examples of typical use-cases. Starting with vertex shaders in section 2.2.1, this section further follows the graphics pipeline to tessellation shaders in section 2.2.2 and geometry shaders in section 2.2.3. Fragment shaders and compute shaders are also covered in sections 2.2.4 and 2.2.5.

2.2.1 Vertex Shader

As seen in Figure 2.1 the vertex shader is the first stage in the graphics pipeline. When talking about vertices in computer graphics, a vertex can be seen as the corner of a triangle where two edges meet. Thus every triangle consists of three vertices and multiple triangles form a mesh to represent 3D objects. Such vertices can be defined by various attributes, like their position, color or texture information. A vertex shader will be invoked exactly once per vertex and can do mathematical operations on these vertex attributes. The simplest implementation of a vertex shader would just transforms the position of a vertex from world space to screen space, but there is no limit to the visual effects that can be created. A programmer could use vertex shaders to do per vertex lighting or do complex deformations of surfaces, hence creating water wave effects or lens effects. Figure 2.2 shows an example of a vertex shader used for displacement mapping.

Figure 2.2: Benefits of displacement mapping using vertex texturing [35].

2.2.2 Tessellation Shader

Tessellation shaders operate on sets of vertices, called patches. In general the task of a tessellation shader is to subdivide these patches into smaller, more refined, primitives according to a mathematical function. As a result tessellation creates more vertices with all needed data, like position, color or texture coordinates. This enables programmers to add more details to polygons in real time. As an easy example a cube could be tessellated to a sphere depending on the distance to the camera. As shown in Figure 2.1 tessellation is split into two shader stages. The first

stage is the tessellation control shader (TCS) stage, also called hull shader stage in Direct3D. A tessellation control shader defines the size of the patches and decides the amount of tessellation each patch gets. It has to ensure that shared edges of patches use the same level of tessellation, to prevent holes in the resulting meshes. After the TCS stage the patches are passed to the tessellator, who actually generates the new primitives. The second shader stage is the tessellation evaluation shader (TES) stage also called Domain Shaders for the Direct3D equivalent. The tessellation evaluation shader takes the patches it got from the TCS stage and calculates the new vertex positions, colors or texture coordinates. While a tessellation control shader can use a default implementation for size of patches and hence is optional, a tessellation evaluation shader is always required to do tessellation. Figure 2.3 compares the rendering of a dragon with and without tessellation.

Figure 2.3: Rendering without (left) and with (right) hardware tessellation [32].

2.2.3 Geometry Shader

The geometry shader stage is logically placed between vertex shader stage, or tessellation shader stage if used, and the fragment shader stage. Geometry shaders can process primitive types as input and generate one or more primitives as output.

These types can be points, lines or triangles. While input and output types do not have to match, a single geometry shader can only output one type of primitive but multiple instances of it. When geometry shaders were introduced, they were also used to implement some crude forms of tessellation, but the performance was far from satisfactory, leading to separate shader stages for tessellation as described above. More natural applications of geometry shaders are layered renderings or the creation of shadow volumes. Geometry shaders excel at tasks where the same primitives have to be displayed multiple times.

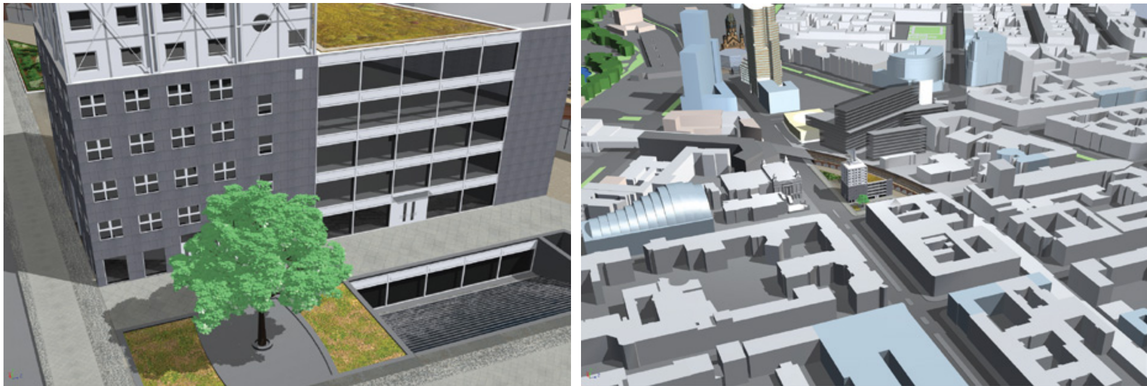


Figure 2.4: Shadow volume generation of complex meshes with a geometry shader. The same scene rendered form different distances [31].

2.2.4 Fragment Shader

After the primitives pass the geometry shader stage clipping and rasterization operations are performed in the rasterization stage. All primitives are traversed and their values get interpolated according to their type. If we think of triangles as a primitive type, the vertices get interpolated across the area of the triangles. The next stage is the fragment shader stage, also known as pixel shaders when using DirectX and HLSL. While the term pixel shader is probably easier to associate with the graphical output of the shader, fragment shader is the more accurate term since it does not directly manipulate pixel colors, but describes how the data of a fragment is used to modify the values of a pixel. While tessellation and geometry shader stages are optional, the vertex shader stage and the fragment shader stage are not. This is the reason why the inputs of fragment shaders often match the outputs of vertex shaders in terms of variables. A fragment shader has one defined output, which is the color

of the processed fragment, but it can still be used to manipulate other data like the generated depth values of the fragments. The graphical effects fragment shaders can produce are almost unlimited, some of the most common use-cases are lighting calculations, shadow creation or bump mapping. Also post-processing effects like blurring or distortion can easily be achieved by using fragment shaders.

Figure 2.5: This figure shows a scene with and without a motion blur effect, created by generating a velocity map in a fragment shader [31].

2.2.5 Compute Shader

As opposed to the other shaders mentioned above, compute shaders are not tied to the rendering pipeline. This is why compute shaders are not shown in the pipeline in Figure 2.1. Compute shaders are mainly used for general-purpose computing tasks on graphics processing units (GPGPU tasks). Although the calculated values can be used to support rendering tasks, compute shaders can process arbitrary information [14]. When heavy computational task can be parallelized, those task can be calculated efficiently on graphics cards, this is exactly what compute shaders are designed for.

2.3 Shading Languages

This section briefly covers the most common shading languages used today. The two most common languages, the OpenGL Shading Language (GLSL) in section 2.3.1 and the High Level Shading Language (HLSL) in section 2.3.2 are discussed first. Followed by a quick look at the Metal Shading Language (MSL) in section 2.3.3. As

the Standard Portable Intermediate Representation (SPIR) is most important for understanding this thesis, section 2.3.4 describes SPIR and also the Vulkan version SPIR-V in more detail.

2.3.1 OpenGL Shading Language (GLSL)

As stated in 2.1 the OpenGL Shading Language, as we know it, was introduced by the OpenGL Architecture Review Board (OpenGL ARB) in 2004 [19]. In Fall of 2006, the control of OpenGL and therby the control of GLSL was transferd to the Khronos Group. GLSL was designed to be a cross-platform language, therefore it runs on every platform supporting the OpenGL API, like Linux, macOS or Windows. For mobile platforms the OpenGL ES Shading Language (OpenGL for Embedded Systems [16]) is available, also known as GLSL ES or ESSL, which supports a great subset of GLSL. Since GLSL is a C-like language it contains the operators known from C or C++ with the exception of pointers, also branching and user defined functions are supported. Due to the architecture of GPUs, recursion is not supported [40].

2.3.2 High Level Shading Language (HLSL)

The High Level Shading Language (HLSL [30]) was released by Microsoft alongside Direct3D 9, the graphical subset of DirectX 9, in 2002. Like GLSL it is a C-like language to create programmable shaders. Unlike GLSL, HLSL supports Windows platforms only, including Windows Mobile and XBox systems. This may sound restricting at first, but not having to deal with compability concerns enables HLSL and also Direct3D to evolve much quicker than GLSL.

2.3.3 Metal Shading Language (MSL)

The Metal Shading Language (MSL [2]) is a shading language designed for the use with Apple's Metal API. Metal was introduced by Apple in 2014 and is a graphics API for creating apps specifically for the Apple platforms macOS, iOS and tvOS. The Metal Shading Language is a C++ based shading language with great support for general-purpose data-parallel computations.

2.3.4 Standard Portable Intermediate Representation (SPIR)

The Standard Portable Intermediate Representation (SPIR [21]) is developed and maintained by the Khronos Group and was originally designed as a mapping from their OpenCL standard to the LLVM [26] intermediate representation. With respect to the Vulkan API, meant to be the successor of OpenGL, SPIR has evolved to a true cross-API with native support for kernel and shader features and now goes with the name SPIR-V. SPIR-V was fully defined by the Khronos Group and is meant to be a simple binary intermediate representation for graphical shaders and compute kernels. It is designed to map very easily to other shading languages, like GLSL or HLSL. With SPIR-V the first steps of compilation and reflection, as well as some optimizations, can be done offline. A SPIR-V binary follows the static single assignment form (SSA form [3]) and is basically a stream of assembler-like opcodes (operation codes). Hence, a unique id (identifier) is assigned to each instruction in the SPIR-V binary. This enables compilers to easily optimize it, but as SPIR-V is a binary format it is not human-readable. Example 2, taken from the SPIR-V specification document [21], tries to display how SPIR-V would look in readable form. The shown shader is the equivalent to the corresponding fragment shader depicted in example 1. A SPIR-V module is a linear stream of words following a well defined layout. As seen in example 2, every binary begins with a `Magic` number, this number can be used as pleased. However, a useful application would be to identify the endianness that is applied for conversion when the module is stored as a stream of bytes in a file. The `Version` variable defines the SPIR-V version, for this thesis Version 1.0.0 was always used. The `Generator` definition gives information about what tool was used to create the SPIR-V binary. In the example below the `Bound` specifies the maximum id used in the shader. It has to be guaranteed that every id is between 0 and `Bound`. Ideally `Bound` should be as small as possible to have densely packed ids near 0. This is also the reason why a remap id pass was introduced as part of this thesis, this will be explained in more detail in later chapters. After the `Schema` setting which is just reserved for now and can bet set to 0, the instruction stream is starting. The first instructions define shader capabilities, memory models, executation modes and the entry point of the shader. The instructions have to be in the exact same order as shown in example 2.

After this general information, all debug information is defined. Debug information contains source extensions, variable names and member names of structs. The debug information is followed by an `Annotations` section. Annotations provide information about type decorations, such as array strides or memory offsets when dealing with arrays. Also precision identifiers or interpolation qualifiers are defined here. SPIR-V defines all types, variables and constants directly after the `Annotations` section. After this definition, the actual shader code starts. In SPIR-V every function starts with an `OpFunction` opcode and ends with an `OpFunctionEnd` opcode. Inside this function the opcodes are grouped into blocks, where each block starts with `OpLabel` and ends with a branch instruction. Branch instructions define the control flow and can either be `OpBranch`, `OpBranchConditional` (if), `OpSwitch` (switch), `OpKill` (termination), `OpReturn` (function return), `OpReturnValue` (function return value) or `OpUnreachable`.

Example 1 The GLSL fragment shader, corresponding to the SPIR-V shader in example 2, as shown in the SPIR-V specification.

```
1:  #version 450

2:  in vec4 color1;
3:  in vec4 multiplier;
4:  noperspective in vec4 color2;
5:  out vec4 color;

6:  struct S {
7:      bool b;
8:      vec4 v[5];
9:      int i;
10: };

11: uniform blockName {
12:     S s;
13:     bool cond;

14: void main()
15: {
16:     vec4 scale = vec4(1.0, 1.0, 2.0, 1.0);

17:     if (cond)
18:         color = color1 + s.v[2];
19:     else
20:         color = sqrt(color2) * scale;

21:     for (int i = 0; i < 4; ++ i)
22:         color *= multiplier;
23: }
```

Example 2 SPIR-V binary example in human-readable form, as shown in the SPIR-V specification.

```
; Magic:     0x07230203 (SPIR-V)
; Version:   0x00010000 (Version: 1.0.0)
; Generator: 0x00080001 (Khronos Glslang Reference Front End; 1)
; Bound:     63
; Schema:    0

               OpCapability Shader
          %1 = OpExtInstImport "GLSL.std.450"
               OpMemoryModel Logical GLSL450
               OpEntryPoint Fragment %4 "main" %31 %33 %42 %57
               OpExecutionMode %4 OriginLowerLeft

; Debug information

               OpSource GLSL 450
               OpName %4 "main"
               OpName %9 "scale"
               OpName %17 "S"
               OpMemberName %17 0 "b"
               OpMemberName %17 1 "v"
               OpMemberName %17 2 "i"
               OpName %18 "blockName"
               OpMemberName %18 0 "s"
               OpMemberName %18 1 "cond"
               OpName %20 ""
               OpName %31 "color"
               OpName %33 "color1"
               OpName %42 "color2"
               OpName %48 "i"
               OpName %57 "multiplier"

; Annotations (non-debug)

               OpDecorate %15 ArrayStride 16
               OpMemberDecorate %17 0 Offset 0
               OpMemberDecorate %17 1 Offset 16
               OpMemberDecorate %17 2 Offset 96
               OpMemberDecorate %18 0 Offset 0
               OpMemberDecorate %18 1 Offset 112
               OpDecorate %18 Block
               OpDecorate %20 DescriptorSet 0
               OpDecorate %42 NoPerspective
```

Example 2 SPIR-V binary example in human-readable form continued.

```
; All types, variables, and constants

 %2 = OpTypeVoid
 %3 = OpTypeFunction %2                          ▷ void()
 %6 = OpTypeFloat 32                             ▷ 32-bit float
 %7 = OpTypeVector %6 4                          ▷ vec4
 %8 = OpTypePointer Function %7                  ▷ function-local vec4*
%10 = OpConstant %6 1
%11 = OpConstant %6 2
%12 = OpConstantComposite %7 %10 %10 %11 %10     ▷ vec4(1,1,2,1)
%13 = OpTypeInt 32 0
%14 = OpConstant %13 5
%15 = OpTypeArray %7 %14
%16 = OpTypeInt 32 1
%17 = OpTypeStruct %13 %15 %16
%18 = OpTypeStruct %17 %13
%19 = OpTypePointer Uniform %18
%20 = OpVariable %19 Uniform
%21 = OpConstant %16 1
%22 = OpTypePointer Uniform %13
%25 = OpTypeBool
%26 = OpConstant %13 0
%30 = OpTypePointer Output %7
%31 = OpVariable %30 Output
%32 = OpTypePointer Input %7
%33 = OpVariable %32 Input
%35 = OpConstant %16 0
%36 = OpConstant %16 2
%37 = OpTypePointer Uniform %7
%42 = OpVariable %32 Input
%47 = OpTypePointer Function %16
%55 = OpConstant %16 4
%57 = OpVariable %32 Input

; All functions

 %4 = OpFunction %2 None %3                      ▷ main()
 %5 = OpLabel
 %9 = OpVariable %8 Function
%48 = OpVariable %47 Function
      OpStore %9 %12
%23 = OpAccessChain %22 %20 %21                  ▷ location of cond
%24 = OpLoad %13 %23                             ▷ load 32-bit int from cond
```

Example 2 SPIR-V binary example in human-readable form continued.

```
%27 = OpINotEqual %25 %24 %26                 ▷ convert to bool
      OpSelectionMerge %29 None               ▷ structured if
      OpBranchConditional %27 %28 %41         ▷ if cond
%28 = OpLabel                                 ▷ then
%34 = OpLoad %7 %33
%38 = OpAccessChain %37 %20 %35 %21 %26       ▷ s.v[2]
%39 = OpLoad %7 %38
%40 = OpFAdd %7 %34 %39
      OpStore %31 %40
      OpBranch %29
%41 = OpLabel                                 ▷ else
%43 = OpLoad %7 %42
%44 = OpExtInst %7 %1 Sqrt %43
%45 = OpLoad %7 %9
%46 = OpFMul %7 %44 %45
      OpStore %31 %46
      OpBranch %29
%29 = OpLabel                                 ▷ endif
      OpStore %48 %35
      OpBranch %49
%49 = OpLabel
      OpLoopMerge %51 %52 None                ▷ structured loop
      OpBranch %53
%53 = OpLabel
%54 = OpLoad %16 %48
%56 = OpSLessThan %25 %54 %55                 ▷ i<4?
      OpBranchConditional %56 %50 %51         ▷ body or break
%50 = OpLabel                                 ▷ body
%58 = OpLoad %7 %57
%59 = OpLoad %7 %31
%60 = OpFMul %7 %59 %58
      OpStore %31 %60
      OpBranch %52
%52 = OpLabel                                 ▷ continue target
%61 = OpLoad %16 %48
%62 = OpIAdd %16 %61 %21                      ▷ ++i
      OpStore %48 %62
      OpBranch %49                            ▷ loop back
%51 = OpLabel                                 ▷ loop merge point
      OpReturn
      OpFunctionEnd
```

2.4 Compiler Optimization

This section discusses some basic compiler optimizations relevant to this thesis. The principals and ideas of these optimizations are pointed out using concrete examples. Dead code elimination in section 2.4.1 and inline expansion in section 2.4.2 will be explained in the beginning, whereas constant propagation and copy propagation will be focused afterwards in sections 2.4.3 and 2.4.4.

2.4.1 Dead Code Elimination

Dead code elimination (DCE) is a very popular compiler optimization and is usually performed on the intermediate representation of source code, which typically is in static single assignment form (SSA form) [42]. Dead code elimination aims to identify and and remove code instructions, whose execution have no impact on the result of a program. This includes instructions that are never reached or variables that are never used. Also blocks of branches, which can be evaluated at compile time, can be considered dead if they will not be part of the control flow. Dead code is not always connected to bad programming it can also be the result of various operations, like other optimization passes or intermediate compilation steps. Considering unused functions and variables dead code elimination can greatly reduce the code size, which itself can lead to better performance of the program due to lower memory consumption and better cache usage [4]. Example 3 shows some types of dead code. Assuming entry point of the program is the `main` procedure, the variable `y` declared in line 6 can be considered dead, because it is never used in the rest of the program. Sometimes conditions can be evaluated during compile time and when taking a look at the if-else construct in the main procedure, it is obvious that the else block will never be executed and thus it can also be considered dead. Since the program in Example 3 will return in line 8 the code in line 12,13 and 14 will not be reached, as a consequence the procedure `INCREMENT` will never be called and also rendered dead.

Example 3 Different types of dead code.

```
1:  procedure INCREMENT(x)                          ▷ Dead function
2:      return x + 1
3:  end procedure

4:  procedure MAIN
5:      x = 1
6:      y = 2                                       ▷ Dead variable
7:      if True then
8:          return x
9:      else
10:         return 0                                ▷ Dead Block
11:     end if
12:     z = 3                                       ▷ Unreachable code
13:     INCREMENT(z)                                ▷ Unreachable code
14:     return 0                                    ▷ Unreachable code
15: end procedure
```

2.4.2 Inline Expansion

In principle the inline expansion optimization is quite simple, it replaces a function call with the body of the called function. The basic idea is to get rid of the overhead a function call is producing. This includes various things like pushing the calls arguments onto the stack and evaluate them or saving live values to registers to actually perform the jump to the function. When returning from a function to the calling function return values need to be copied leading to allocation and deallocation overheads. Inline expansion is a trade-off between faster execution times and code size. Considering functions with a lot of instructions, inlining them multiple times can considerably bloat the code leading to worse performance. On the other hand inlining functions with only a few instructions can significantly boost performance. Deciding which functions can be effectively inlined is a non-trivial task and some functions cannot be inlined at all [6]. Modern compilers enable programmers to hint at functions that should be inlined with an *INLINE* keyword, but as it is only a hint it is up to the compiler to decide in the end. Example 4 shows a simple code fragment where inline expansion can be used effectively, it compares the code before and after inline expansion.

Example 4 A code example before and after inline expansion.

Before inline expansion:

```
 1: procedure MULTIPLY(a, b)                          ▷ Function to inline
 2:     return a * b
 3: end procedure

 4: procedure MAIN
 5:     x = 1
 6:     y = 2
 7:     z = 3
 8:     result = 0
 9:     if x ≤ y then
10:         result = MULTIPLY(x,z)                    ▷ Call to inline
11:     else
12:         result = MULTIPLY(y,z)                    ▷ Call to inline
13:     end if
14:     return result
15: end procedure
```

After inline expansion:

```
 1: procedure MAIN
 2:     x = 1
 3:     y = 2
 4:     z = 3
 5:     result = 0
 6:     if x ≤ y then
 7:         result = x * z                ▷ Inlined version of MULTIPLY
 8:     else
 9:         result = y * z                ▷ Inlined version of MULTIPLY
10:     end if
11:     return result
12: end procedure
```

2.4.3 Constant Propagation

The constant propagation optimization is often combined with the constant folding optimization. When doing constant folding, the compiler simply replaces every calculation done with constant values only, with the result of this calculation. This leads to less instructions that need to be processed. Constant propagation is related to constant folding, but deals with constant values assigned to variables. If a variable holds a constant value, constant propagation replaces all uses of that variable with the constant value it has assigned [42]. Leading to greatly simplified code, constant propagation is often followed by dead code elimination to clean up unused variables and instructions afterwards. Constant propagation and also constant folding can be performed on various data types, depending on the compiler even string literals can be propagated and folded [33]. Example 5 shows how constant propagation and constant folding can be applied in multiple passes. After propagating the constants `x` and `y`, dead code elimination can be used to reduce the the code even further.

Example 5 A code example before and after constant propagation.

Before constant propagation:

1: **procedure** MAIN
2: ***const*** $x = 10$
3: ***const*** $y = 20 + 18 * x$
4: **return** $30 - 2 * x + y/4$
5: **end procedure**

After propagation of x:

1: **procedure** MAIN
2: ***const*** $x = 10$
3: ***const*** $y = 20 + 18 * 10$ ▷ Constant folding reduces this to $y = 200$
4: **return** $30 - 2 * 10 + y/4$
5: **end procedure**

After propagation of y:

1: **procedure** MAIN
2: ***const*** $x = 10$
3: ***const*** $y = 200$
4: **return** $30 - 2 * 10 + 200/4$ ▷ Constant folding reduces this to $result = 60$
5: **end procedure**

2.4.4 Copy Propagation

The goal of copy propagation is to reduce the number of direct assignments in the source code. Direct assignments are instructions that copy a value from one variable to another and have a form similar to `x = y`. Sometimes such statements can be eliminated completely and thus reduce code size and the number of variables used [33]. As with constant propagation, this optimization is often followed by dead code elimination for cleanup. In Example 6 the reduction of code, due to the propagation of `x` can be observed.

Example 6 A code example before and after copy propagation.

Before copy propagation:

```
1: procedure MAIN
2:     x = RANDOMVALUE()
3:     y = x                              ▷ Direct assignment
4:     z = y * 100
5:     return z
6: end procedure
```

After copy propagation:

```
1: procedure MAIN
2:     x = RANDOMVALUE()
3:     z = x * 100
4:     return z
5: end procedure
```

Chapter 3

SPIR-V Tools

This chapter describes the most important tools available when working with SPIR-V binary shaders. Since SPIR-V is still a young format there are not many alternatives published yet. The sections 3.1, 3.2 and 3.3 cover the official open source tools published by the Khronos Group, while the sections 3.4 and 3.5 deal with third party applications. Section 3.6 addresses SMOL-V, which focuses on optimizing the binary size of SPIR-V modules.

3.1 GLSL Reference Compiler

The GLSL Reference Compiler, also know as just glslang, is the official reference compiler front-end for the OpenGL and also OpenGL ES shading languages [17][18]. It is published under the BSD (Berkeley Software Distribution) license and is maintained by the Khronos Group. The glslang project consists of serveral components. First a GLSL/ESSL front-end, which can validate and translate GLSL into an abstract syntax tree (AST). The second component is an HLSL front-end to translate HLSL into the same AST. At the time of writing this thesis, this component, in contrast to the GLSL front-end, is far from feature complete and can be considered experimental. The next component is a SPIR-V back-end to convert the created AST to SPIR-V. The GLSL Reference Compiler also provides a validation tool for SPIR-V, to always check the created SPIR-V shader against the current specification of the language. Since SPIR-V is a pure binary language, and thus is not directly readable by programmers, debugging a SPIR-V binary is hard to do. Glslang pro-

vides an option to print a readable form of the SPIR-V binary shader, to simplify such tasks. Example 8 shows a simple shader, written in ESSL, which just calculates a vertex position. This shader was then converted to SPIR-V using glslang. The human-readable output is displayed in Exmaple 7. As already mentioned in Section 2.3.4, a SPIR-V binary follows the SSA form. In Example 7 the numbers on the left side represent the unique identifiers of each instruction, while on the right side the opcodes of the instructions are displayed. In SPIR-V the built-in variable `gl_Position` is part of the `gl_PerVertex` struct, this is the reason why `gl_Position` is not found in the SPIR-V representation.

Example 7 Simple Vertex Shader in readable SPIR-V, converted by glslang.

```
2 :                        TypeVoid                        ▷ Start of variable declarations
3 :                        TypeFunction 2
6 :                        TypeFloat 32
7 :                        TypeVector 6(float) 4
8(gl_PerVertex):           TypeStruct 7(fvec4) 6(float)
9 :                        TypePointer Output 8(gl_PerVertex)
10 :              9(ptr)   Variable Output
11 :                       TypeInt 32 1
12 :             11(int)   Constant 0
13 :                       TypeMatrix 7(fvec4) 4
14(UBO):                   TypeStruct 13
15 :                       TypePointer Uniform 14(UBO)
16 :             15(ptr)   Variable Uniform
17 :                       TypePointer Uniform 13
20 :                       TypePointer Input 7(fvec4)
21(aVertex):     20(ptr)   Variable Input
24 :                       TypePointer Output 7(fvec4)

4(main):               2   Function None 3                 ▷ Start of main function
5 :                        Label
18 :             17(ptr)   AccessChain 16 12
19 :                 13    Load 18
22 :            7(fvec4)   Load 21(aVertex)
23 :            7(fvec4)   MatrixTimesVector 19 22
25 :             24(ptr)   AccessChain 10 12
                           Store 25 23
                           Return
                           FunctionEnd
```

Example 8 Simple Vertex Shader in ESSL, before conversion with glslang.

```
1: layout(std140) uniform UBO
2: {
3:     uniform mat4 uMVP;
4: }

5: in vec4 aVertex;

6: void main()
7: {
8:     gl_Position = uMVP * aVertex;
9: }
```

3.2 SPIRV-Cross

While glslang can create SPIR-V code from high level shading languages, SPIRV-Cross can be seen as the direct counterpart to it. The purpose of SPIRV-Cross is to translate from the intermediate language back to various high level languages. It is also maintained by the Khronos Group and can cross-compile SPIR-V into GLSL, MSL, HLSL and even debuggable C++ [20]. SPIRV-Cross has a strong focus on creating well formatted and readable outputs, ideally the created code looks like it was written by hand. The translation to GLSL is already pretty far developed and can be considered mostly feature complete. In contrast to GLSL the support for HLSL, MSL and C++ is limited at the time of writing this thesis. While the translation to those languages does work for simple shaders, the output is far from being as clean and efficient as the GLSL output. As this tool can be used to cross-compile between high level shading languages, for example when a HLSL shader was translated to SPIR-V and is now translated to GLSL by SPIRV-Cross, one has to consider that to target language probably lacks native support for some features used in the origin language. SPIRV-Cross tries to provide tools, like reflection APIs, to handle these scenarios in a robust way and keep the manual actions required to maintain compatibility to a minimum. SPIRV-Cross runs on Linux, macOS and Windows.

3.3 SPIR-V Tools

The SPIR-V Tools project, is actually a collection of library functions helping to process SPIR-V binaries. It consists of an assembler, a disassembler, a binary module parser, a validator and an optimizer for SPIR-V [22]. Similar to the tools mentioned above, the Khronos Groups maintains this project. The assembler in SPIR-V Tools is able to output a SPIR-V binary from an assembly language text, while also doing basic syntax checking. As for all the library functions in this project, there is a standalone command line tool that wraps around the assembler library function, for easier usage. SPIR-V Tools also provides a file to enable syntax highlighting for the SPIR-V assembly language, which can be created by the disassembler, when using the Vim editor. This project comes with a validator library function, which checks the validation rules described in the SPIR-V specification and is supposed to be used to double-check SPIR-V binaries after modifying them. Additionally SPIR-V Tools can also export a visualization of the control flow graph of a SPIR-V binary, using the graph visualization software GraphViz and thus the *DOT* format [12]. Example 9 shows a `main` function of a GLSL shader and Example 10 displays the corresponding SPIR-V binary in human-readable form. Figure 3.1 illustrates the control flow graph SPIRV-Tools printed from the SPIR-V binary using GraphViz. The optimizer provided by the SPIR-V Tools project, is still under development. At the time of writing, the optimizer supports only a few basic optimization passes. Most of the passes are dealing with constants or specialized constants, like folding, freezing, unifying these constants. Also dead constants and debug information can be removed. In terms of dead code elimination, SPIR-V Tools only checks if the resulting value (and the unique id) of an instruction is used. If it is used again in the instruction is kept, otherwise it will be removed. This approach only covers a small part of dead code elimination as explained in section 2.4.1.

3.4 Krafix

Krafix is used in a ultra-portable, high performance and open source multimedia framework called Kha, developed by KTX Software [23]. It wraps around the GLSL Reference Compiler and provides tools to translate the SPIR-V binary to multiple other languages. While it does not do any optimizations, it is able to compile

Example 9 The GLSL shader corresponding to the control flow graph in 3.1.

```
1:  void main()
2:  {
3:      int x = 30
4:      if (x < 15)
5:      {
6:          gl_Position = vec4(1.0);
7:      }
8:      else
9:      {
10:         gl_Position = vec4(0.0);
11:     }
12:     gl_Position+ = vec4(1.0);
13: }
```

Example 10 The SPIR-V shader corresponding to the control flow graph in 3.1.

```
4(main):            2 Function None 3                  ▷ Start of main function
5 :                   Label
8(x):          7(ptr) Variable Function
                      Store 8(x) 9
10 :           6(int) Load 8                           ▷ x = 30
13 :         12(bool) SLessThan 10 11
                      SelectionMerge 15 None
                      BranchConditional 13 14 26       ▷ if(x < 15)
14 :                      Label
25 :              24(ptr) AccessChain 20 21
                          Store 25 23
                          Branch 15
26 :                      Label                        ▷ else
29 :              24(ptr) AccessChain 20 21
                          Store 29 28
                          Branch 15
15 :                   Label
30 :           24(ptr) AccessChain 20 21
                       Store 25 23
33 :           24(ptr) AccessChain 20 21
31 :         17(fvec4) Load 30
32 :         17(fvec4) FAdd 31 23
                       Store 33 32
                       Return
                       FunctionEnd
```

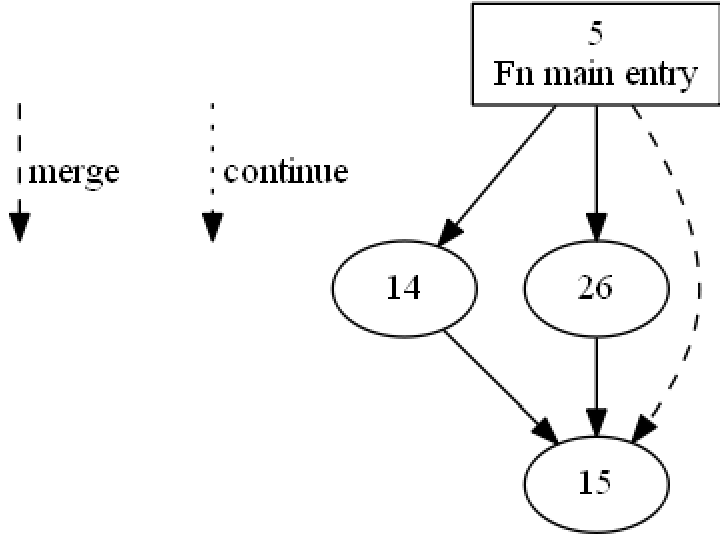


Figure 3.1: Control flow graph example using SPIR-V Tools and GraphViz.

SPIR-V to GLSL, HLSL, MSL and even the Adobe Graphics Assembly Language (AGAL) [24]. The SPIR-V to HLSL conversion, is implemented in two ways. The first approach uses its own implementation and the second one makes use of the SPIRV-Cross framework. This is because Robert Konrad, who developed Krafix, is also working on the SPIR-V to HLSL conversion for SPIRV-Cross. Thus SPIRV-Cross actually uses ported code from Krafix for its HLSL part.

3.5 ShaderC

ShaderC is a collection of tools, libraries and tests for shader compilation. It is owned by Google, but it is not an official Google product, although parts the ShaderC code are shipped wit the Android NDK (Native Development Kit) [11]. ShaderC wraps around the GLSL Reference Compiler and SPIR-V Tools. It provides a command line compiler with GCC- and CLang-like usage for compiling GLSL and HLSL to SPIR-V. It also provides a library which can do the same. It is designed to be an API where new functionality can easily be added and backwards compatibility is guaranteed. The ShaderC compiler extends the functionality of GLSL by enabling the *GL_GOOGLE_include_directive*, allowing shaders to conveniently use *#include* statements.

3.6 SMOL-V

SMOL-V aims to reduce the binary file size of a SPIR-V shader. It was created by Aras Pranckevičius, who also developed the GLSL Optimizer used by the Murl Engine [36][37]. Since SPIR-V is a verbose language in SSA form, its binary file size is several times large than the binaries of other shading languages. SMOL-V allows encoding and decoding of SPIR-V binaries without changing their behavior. It optimizes the usage of the unique identifiers, resulting from the SSA form, in a way that identifiers which are used more frequently will have lower numbers. SMOL-V also swaps values of the opcodes SPIR-V provides, dependent on how often they are used on average. While an `OpDecoration` opcode will be used multiple times in each SPIR-V module, `OpMemoryModel` will only be used once. To lower the file size it makes sense to swap such opcodes and save a few bits. After optimizing the occurrences of the opcodes, SMOL-V applies a varint encoding (variable-length integer encoding [41]). This encodes integers in a way, that lower numbers can also be stored using less memory. Every number below the value of 128 only takes one byte, while numbers below 16384 take two bytes and so on. With these techniques SMOL-V is able to compress the binary file size a SPIR-V binary to 35% of its original size.

Chapter 4

Design & Implementation

This chapter discusses the design and implementation of the application created during this work, further referred to as SPIR-V Wrapper, in detail. Section 4.1 explains the general design and setup of the application. Afterwards this chapter will elucidate the definition of the interface provided to the Murl Engine for translating and optimizing shading languages. This is followed by a break down of the algorithms used to create the optimizations developed during this work in section 4.2.

4.1 SPIR-V Wrapper

As already mentioned in section 1.1 and displayed in figure 1.4 this application aims to translate GLSL or HLSL to SPIR-V, then perform optimizations on the SPIR-V binary and finally translate the binary back to GLSL or HLSL. While the SPIR-V binary itself can be directly used as a shader when using the Vulkan API, the result after the backwards translation to GLSL or HLSL is what will be used by the Murl Engine for now. Chapter 3 discussed most of the tools currently available to developers when working with SPIR-V binaries and since those tools will most likely be maintained and supported in the future, it makes sense to use them for the purpose of this application. This approach wraps around the functionalities of the GLSL Reference Compiler for converting high level languages to SPIR-V, SPIRV-Tools to perform optimization and validation and SPIR-V Cross to translate the optimized SPIR-V binary back to another high level language. This setup is

illustrated in figure 4.1.

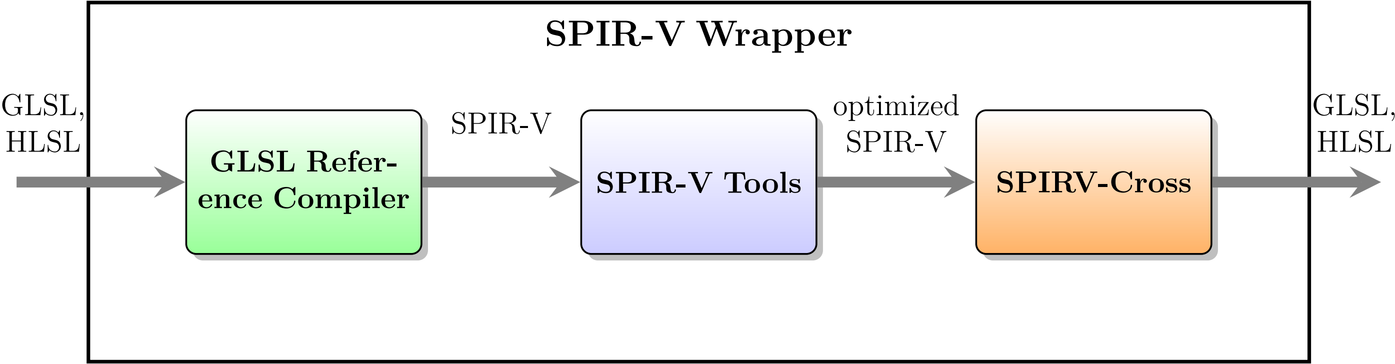


Figure 4.1: The basic setup of the SPIR-V Wrapper application. Provided with a GLSL or HLSL input shader, the wrapper uses glslang as a front-end, SPIR-V Tools for optimization and SPIRV-Cross as a back-end.

With this approach the front-end and the back-end of the application are already well defined, since they are officially supported by the Khronos Group. However, the Murl Engine strongly emphasizes performance and unfortunately the SPIR-V Tools framework can only perform a few minor optimizations on SPIR-V binaries. When providing a high performance cross-platform multimedia framework and especially a game engine, one usually wants to ship the engine with a library of shaders. Such shaders are often shipped as so called uber-shaders, which implement a lot of different functionalities and let the users enable or disable them on demand. If these uber-shaders perform well, strongly depends on how well compilers handle basic optimizations, like dead code elimination or function inlining. The mobile market is flooded with different devices and hence various compilers. To be less reliant on their implementation the Murl Engine is using the GLSL Optimizer to do some offline optimizations before the shader code gets fed to the compiler. To not lose this performance gain, the SPIR-V Wrapper also needs to do similar optimizations on the SPIR-V binary. This is why multiple optimization passes were added to the SPIR-V Tools framework during this work. The added optimization passes include further dead code elimination, inline expansion, copy propagation and id remapping and are described in detail in section 4.2. Figure 4.2 shows the optimization passes already implemented by SPIR-V Tools and the passes added during this work.

The SPIR-V Wrapper application, is meant to be integrated to the Murl Engine as a third-party tool. To ease up integration into the build systems, SPIR-V Wrapper

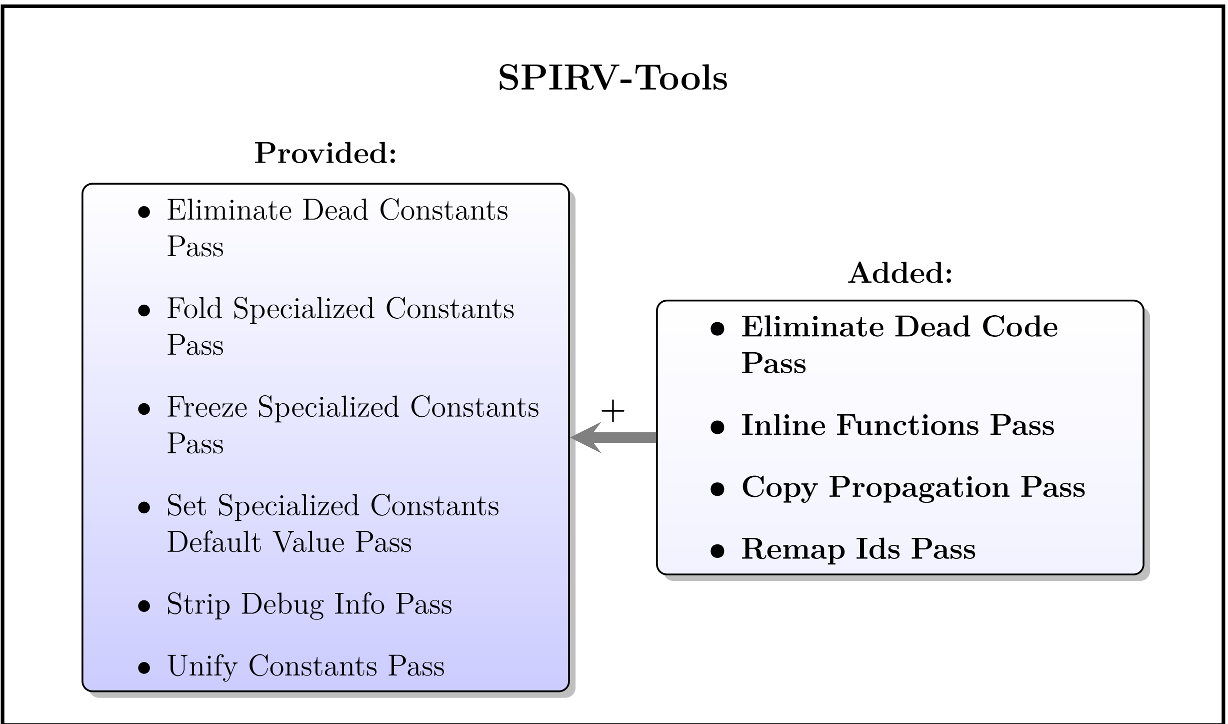


Figure 4.2: This figure shows the optimization passes provided by the SPIR-V Tools framework and the passes that were added during this work.

can be used directly from the command-line as a standalone application, this is explained in 4.1.1. It also provides a simple and easy to use C-interface which is further described in section 4.1.2.

4.1.1 Command Line Usage

As already mentioned above, SPIR-V Wrapper can be used from the command line. This section, shows which command line arguments are supported and describes their usage. From the command line the application can be executed with the following command:

```
SPIRV-Wrapper [option]... [file]...
```

Here file represents an input file, like a vertex shader or a fragment shader that shall be converted. One has to be careful that the file has the correct file ending, since the GLSL Reference Compiler relies on correct file endings to detect the shader type. The supported file endings for the corresponding shader stages are listed below.

Shader stage	**File ending**
Vertex Shader	`.vert`
Tessellation Control Shader	`.tesc`
Tessellation Evaluation Shader	`.tese`
Geometry Shader	`.geom`
Fragment Shader	`.frag`
Compute Shader	`.comp`

Table 4.1: This table shows the supported shader stages and their corresponding file endings, as used by the SPIR-V Wrapper.

Without any further arguments defined, besides the input file, SPIR-V Wrapper will try to convert the provided input shader to SPIR-V, but it will not produce any output. To control what actually happens with the converted shader, SPIR-V Wrapper can be called with the arguments shown in table 4.2. The options `--glsl` and `--hlsl` specify, if the input shader shall be converted to GLSL or HLSL. With `--spv` set, the SPIR-V binary is stored separately in a *.spv* file. In case this is combined with `-o` SPIR-V Wrapper will also output the optimized SPIR-V binary file with *_opt* added to it's filename. The human-readable version seen in section 3.1 and section 3.3 can be created with the `-h` option, this will also extend the filename of the SPIR-V binary by *_readable*. The entry point option is only used if the input shader is written in HLSL. Since GLSL shaders always start with a `main` function, this is the default value, but HLSL shaders can have custom entry points, which need to be specified if used. With the provided option `--watchpoint` the user can inject a watchpoint at a specific position in the source code. This is done by stating the line and the column where the watchpoint should be inserted, e.g. `3:2`. The first number defines the row and the second number defines the column in the input file. Watchpoints are explained in more detail in section 4.3.

4.1.2 Interface Definition

SPIR-V Wrapper provides a C++-interface and a C-interface, for backwards compatibility with other C frameworks. The C-interface is visualized in figure 4.3. When using the interface, `constructWrapper()` has to be called first to create a `SpvwHandle` to work with. This handle has to be destroyed with `destructWrapper()` when it is no longer needed, to clean up memory.

Option	Description
`-h`	Output human-readable form of SPIR-V to file
`-o`	Enable optimizations
`--glsl`	Specify GLSL output file
`--hlsl`	Specify HLSL output file
`--spv`	Specify SPIR-V binary output file
`--entrypoint`	Specify shader entry point, default is `main`
`--watchpoint`	Specify watchpoint position

Table 4.2: This table shows which command line arguments are handled by the SPIR-V Wrapper application.

The typical workflow when using SPIR-V Wrapper is to call `convertToSpv()` followed by `optimizeSpv()` and afterwards either use `convertToGLSL()` or `convertToHLSL()`. As the name suggests, `convertToSpv()` converts the shader code defined by `sourceFile` to a SPIR-V binary, which is the optimized calling `optimizeSpv()`. The optimization passes that should be applied need to be passed using the `optPasses` array. The `optPasses` array is allowed to contain multiple instances of the the same optimization pass to support multi-pass optimizations. SPIR-V Wrapper will run the optimization passes in the exact same order they are defined in this array. The different types of passes available are listed in table 4.3. The two methods `convertToGLSL()` and `convertToHLSL()` can then further be used to convert the SPIR-V binary back to the high level language desired. SPIR-V Wrapper automatically detects shader types, version numbers and profiles.

OptPassTypes
`OptPassTypeNone`
`OptPassTypeStripDebugInfo`
`OptPassTypeUnifyConstant`
`OptPassTypeFoldSpecConstantOpAndComposite`
`OptPassTypeEliminateDeadConstant`
`OptPassTypeEliminateDeadCode`
`OptPassTypeInlineFunctions`
`OptPassTypeCopyPropagation`
`OptPassTypeRemapIds`

Table 4.3: This table lists optimization passes supported by SPIR-V Wrapper.

C-Interface

≪interface≫
SpvwIWrapper

```
constructWrapper() : SpvwHandle
destructWrapper(handle : SpvwHandle) : void

convertToSpv(handle : const SpvwHandle,
             sourceFile : const char*,
             destinationFile : const char*,
             sourceEntryPoint : const char*,
             outputSpv : bool,
             outputReadableSpv : bool,
             watchPointLine : int,
             watchPointColumn : int) : bool

optimizeSpv(handle : const SpvwHandle,
            optPasses : OptPassType[],
            numberOfOptPasses : uint,
            destinationFile : const char*,
            outputSpv : bool,
            outputReadableSpv: bool) : bool

convertToGLSL(handle : const SpvwHandle, destinationFile : const char*) : bool
convertToHLSL(handle : const SpvwHandle, destinationFile : const char*) : bool
```

Figure 4.3: The C-interface provided by the SPIR-V Wrapper application.

4.2 SPIR-V Optimization

This section describes the used algorithms and the implementation of the optimization passes used to extend the SPIR-V Tools framework in detail. To keep the conflicts with future updates of SPIR-V Tools to a minimum, the passes added during this work mostly add functionality, without changing the existing code in SPIR-V Tools project.

4.2.1 Dead Code Elimination Pass

As dead code elimination is one of the most important optimizations, this is the first one this thesis will cover. The dead code elimination pass introduced with the SPIR-V Wrapper uses a mark and sweep approach. The idea is to split the dead code elimination pass into two major parts, the first part will parse through all the instructions and tries to identify unused or never executed code blocks. Those blocks will be marked as dead and will be removed in the sweep part which follows afterwards [47]. While the SPIR-V Tools project already provides methods to analyze instructions and also groups the instructions into blocks according to the SPIR-V specification, the information about how the blocks are related to each other in regard of the control flow is missing. To make up for this, the algorithm used during this work parses through the instructions and the blocks. For each block it remembers the parent block, all the child blocks, as well as the merge instructions and branch instructions used in the block. To conveniently access this data a helper class was introduced, called `BasicBlockInfo`, this is explained further in section 4.2.5.1. After gathering the information about parent and child relations, the dead code elimination pass tries to evaluate conditional branch statements like *if-else* constructs, *switch* instructions and loop conditions. If any condition can be evaluated to true or false during compile time, the dead code elimination pass can mark the block that will never be executed, along with all its children recursively. After stepping through all the conditions, the algorithm can move on to the sweep part and eliminate these blocks. When removing the uncalled blocks, it is crucial to make sure that the control flow of the SPIR-V binary is still valid according to the specification. As an example, if the *else* clause of an *if-else* construct is removed, the *if* instruction can be removed entirely, since there is no longer a branch needed. Also the branch instruction of the remaining block may needs to be corrected. While this is straightforward for simple branches, complexity is much higher for *switch* instructions and loops. The algorithm used during this work only removes loops if the loop condition will never be met. For *switch* instructions there are multiple scenarios to consider. A switch instruction is usually followed by multiple *cases* which have to be evaluated, if the dead code elimination pass is able to evaluate which case will be used, the entire switch can usually be marked as dead and only the one case is kept. There are exceptions though, like when there is no *break* statement used in a

case, the following case also needs to be kept alive. When the removal of the dead blocks is done and all branches and merge constructs are corrected to ensure valid control flow, the dead code elimination pass looks for single instructions that are unused and can also be removed. This often happens as a result of removing entire code blocks as described. SPIR-V Wrapper first checks for local variables that are no longer needed, also considering SPIR-V *access chain* instructions, representing calls to arrays or structs. Since SPIR-V binaries follow the static single assignment form, every instruction results in a unique result id, this allows easy checking if an instruction's result is used later in the code or not. The dead code elimination pass removes every instruction that has a result id that is never used in any other instruction. An exception for this are function calls, these are handled separately, since functions will most likely have side effects, even if their return value is not further used. The algorithm used during this work, considers function calls as dead, if the function does not write to any global variable and it does not manipulate variables passed to it using the *out* or *inout* qualifier. Also the return value of the function, if there is one, must not be used. If these conditions match, the algorithm removes the function call and also the complete function along with its definition, if this was the only call to that function. The dead code elimination pass further checks for unnecessary store instructions. This addresses multiple subsequent stores to the same variable, where the first store gets overridden by the second one. With all these instructions removed, the SPIR-V binary can now contain empty code blocks and unused decorations, these are cleaned up at the end of the pass. Figure 4.4 shows the control flow of the dead code elimination pass as implemented in the SPIR-V Wrapper application and figure 4.5 provides an overview of the dead code elimination pass class.

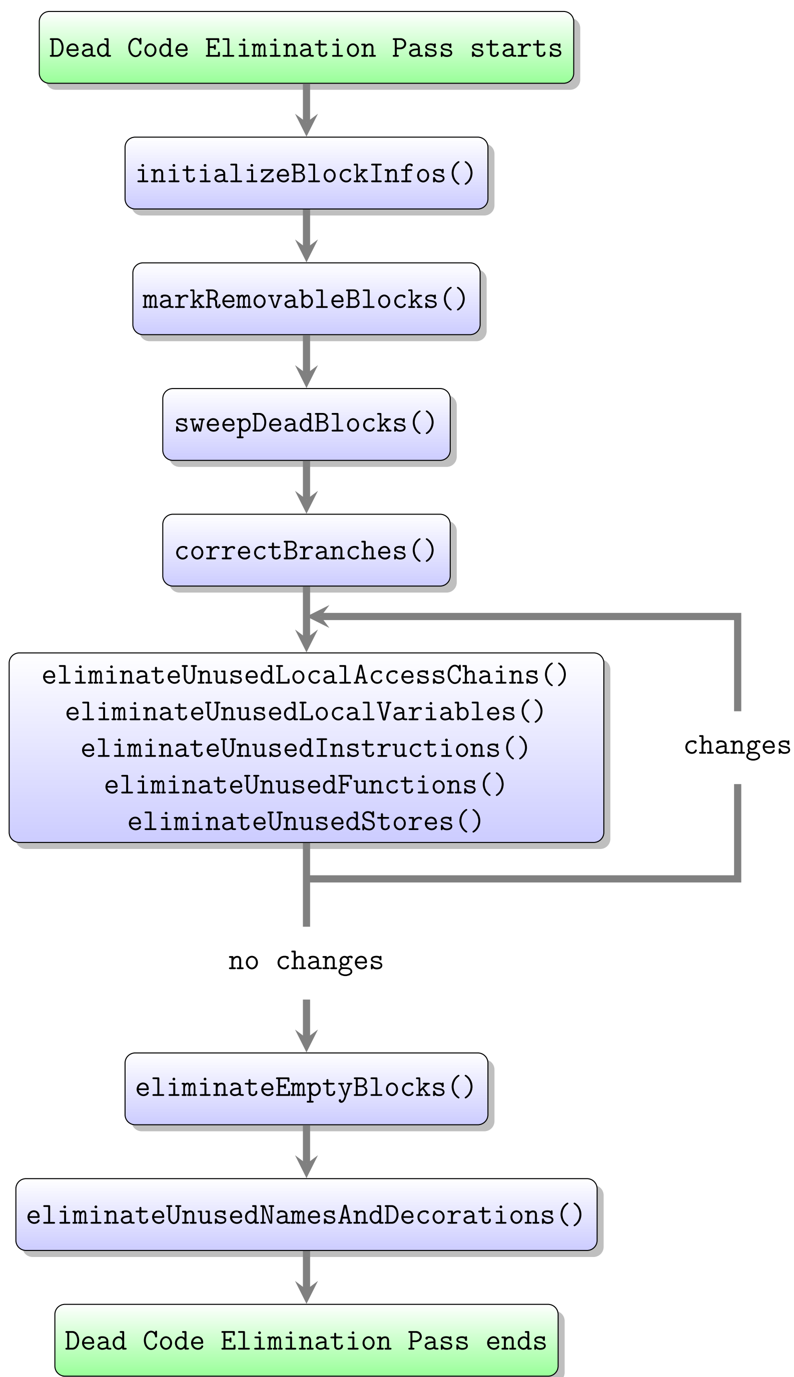


Figure 4.4: This figure shows the basic control flow of the dead code elimation pass, implemented during this work.

Optimization Passes

DeadCodeEliminationPass

private:
mModule : ModulePtr
mDefUseMgr : DefUseMgrPtr
mBlockInfos : BasicBlockInfoMap
mBlocksToRemove : Vector<uint>

public:
Process(module : ModulePtr) : Status

private:
initializeBlockInfos() : void
identifyRemoveableBlocks() : void
eliminateDeadBlocks() : bool
correctBranches() : bool
eliminateUnusedLocalVariables() : bool
eliminateUnusedLocalAccessChains() : bool
eliminateUnusedInstructions() : bool
eliminateUnusedFunctions() : bool
eliminateUnusedStores() : bool
eliminateUnusedBlocks() : bool
eliminateUnusedNamesAndDecorations() : bool

evaluateConditionalBranch(blockInfo : BasicBlockInfoPtr, inst : InstructionPtr) : void
evaluateSwitch(blockInfo : BasicBlockInfoPtr, inst : InstructionPtr) : void

correctConditionalBranch(blockInfo : BasicBlockInfoPtr, inst : InstructionPtr) : bool
correctSwitch(blockInfo : BasicBlockInfoPtr, inst : InstructionPtr) : bool
correctBranch(blockInfo : BasicBlockInfoPtr, inst : InstructionPtr) : bool

handleMergeInstruction(blockInfo : BasicBlockInfoPtr, inst : InstructionPtr) : void
handleBranchInstruction(blockInfo : BasicBlockInfoPtr, inst : InstructionPtr) : void

getParentBlock(childBlock : InstructionPtr) : InstructionPtr
getBasicBlockToAddInstruction(blockInfo : BasicBlockInfoPtr) : BasicBlockInfoPtr
isPartOfLoopMergeInstruction(instructionId : uint, ignoreContinue : bool) : bool
isPartOfSelectionMergeInstruction(instructionId : uint) : bool
isInsideALoop(blockInfo : BasicBlockInfoPtr) : bool

isUniqueBranchInstruction(branchInst : InstructionPtr) : bool
markBlockAndChildren(blockInfo : BasicBlockInfoPtr) : void
isBlockEmpty(labelInst : InstructionPtr) : bool
isDeadInst(inst : InstructionPtr) : bool

Figure 4.5: Dead code elimination pass class.

4.2.2 Inline Functions Pass

The next optimization pass that was implemented, is the inline functions pass. The aim of this pass is to remove the overhead of function calls by replacing every function call with the actual body of the called function. As Hwu et. al. [29] state, the order in which the function calls are inlined is crucial. A different inlining order also results in different outcomes and choosing the right order can reduce the instructions that need to be copied significantly. With this in mind, the algorithm used in the SPIR-V Wrapper first needs to build a call tree to know which functions get called from which function. As with the dead code elimination pass, the inline function pass needs this additional information on top of the data provided by the SPIR-V Tools framework, to build this graph. Hence, there was also a helper class introduced called `FunctionInfo` which is explained in more detail in section 4.2.5.2. After parsing through the shader code and initializing the call tree, the algorithm needs to select which function to inline first. As suggested by Hwu et. al. [29] the algorithm starts inlining the leafs of the call tree and works its way up to the root function. The algorithm created during this work takes a chosen function, identifies all function call instructions used by it and copies the instructions of the called function to the calling function. For all the local variables in the called function, as well as the function parameters and the return value, the algorithm creates a local variable in the calling function. These local variables are mapped to the actual function call, this guarantees that multiple calls to the same function can be handled. The body of the called function is then copied instruction by instruction, replacing the used variables with the newly created ones. After doing this for all the function calls in the shader, there should only be the `main` function left with no function calls. The inline functions pass then cleans up the now obsolete function definitions. Figure 4.6 shows the control flow of the inline functions pass as implemented in the SPIR-V Wrapper and figure 4.7 provides an overview of the inline functions pass class.

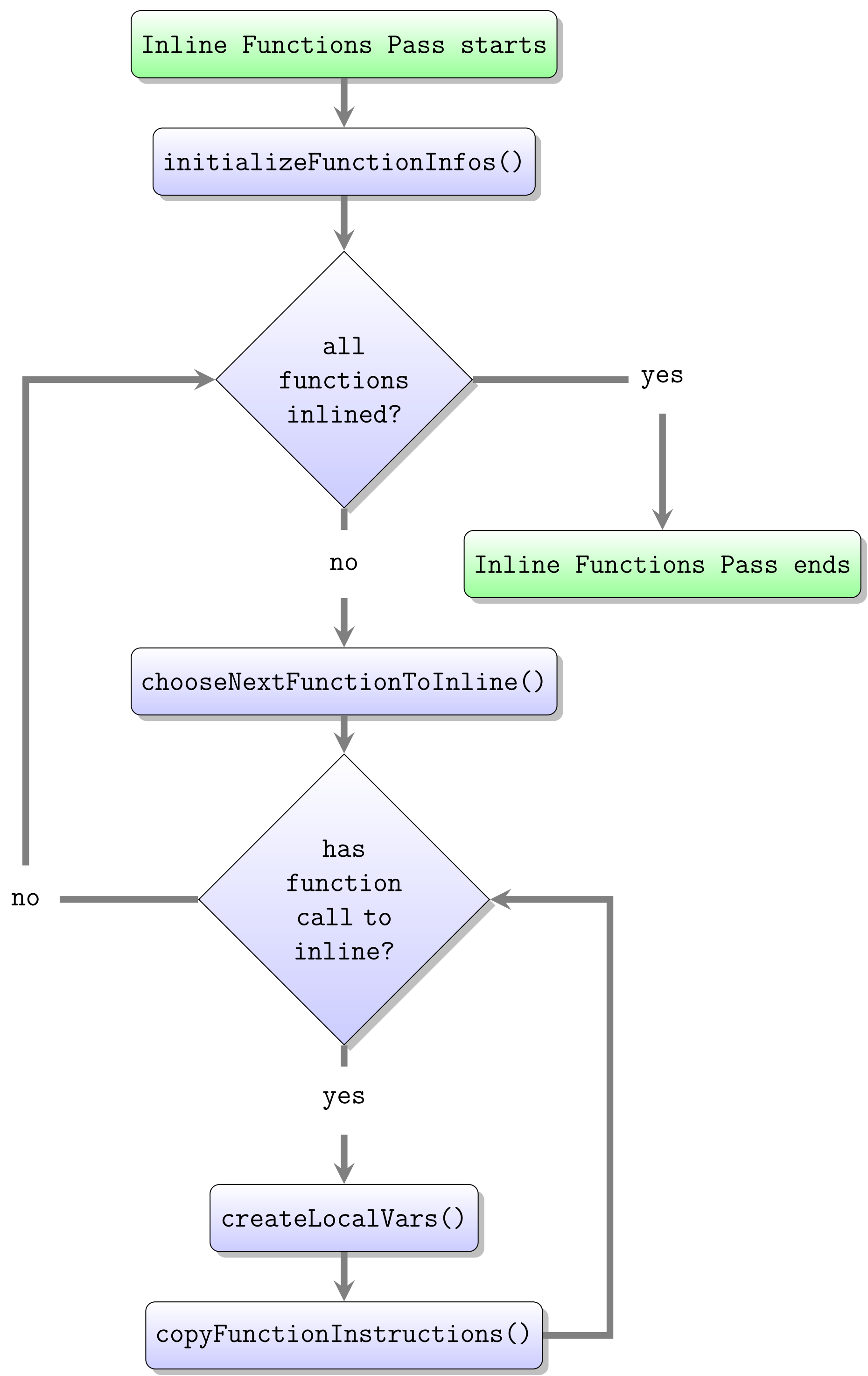


Figure 4.6: This figure shows the basic control flow of the inline function pass.

Optimization Passes

InlineFunctionsPass

private:
mModule : ModulePtr
mDefUseMgr : DefUseMgrPtr
mFunctionInfos : FunctionInfoMap
mFunctionToIdMap : Map<FunctionPtr, int>
mResultIdOffset : uint

public:
Process(module : ModulePtr) : Status

private:
initializeFunctionInfos() : void
getNextFunctionToInline() : FunctionPtr
inlineFunction(function : FunctionPtr) : bool
createInlinedFunction(function : FunctionPtr,
    functionLabel : InstructionPtr,
    callResultIdToFunctionId : Map<uint, uint>,
    callResultIdToParams : Map<uint, Vector<uint>>,
    functionIdToReturnInst : Map<uint, InstructionPtr>,
    functionIdToFunctionTypeId : Map<uint,uint>,
    functionIdToLocalVars : Map<uint, Vector<InstructionPtr>>,
    functionIdToParams : Map<uint, Vector<InstructionPtr>>) : bool

cleanUpFunctions() : bool
getNextFreeResultId() : uint
getFunctionInfo(function : FunctionPtr) : FunctionInfoPtr
getFunctionInfo(functionId : uint) : FunctionInfoPtr
printInliningStatus() : void
killFunction(function : FunctionPtr) : void

createFunctionVariableAndType(typeId : uint) : InstructionPtr
copyLocalVarInstruction(inst : InstructionPtr,
    oldToNewIds : Map<uint, uint>) : InstructionPtr
copyInstructionAndOperands(inst : InstructionPtr,
    oldToNewIds : Map<uint, uint>) : InstructionPtr
copyFunctionBody(blocks : Vector<BasicBlockPtr>,
    calledFunction : FunctionPtr,
    callingFunction : FunctionPtr,
    calledFunctionTypeId : uint,
    functionReturnVar : InstructionPtr,
    functionCallResultId : uint,
    oldToNewIds : Map<uint, uint>) : void

Figure 4.7: Inline functions pass class.

4.2.3 Copy Propagation Pass

As already mentioned in section 2.4.4 the goal of the copy propagation optimization is to reduce the number of direct assignments, to prevent unnecessary copying of variables. Direct assignments in most languages have a form similar to `x = y`. These direct assignments need to be identified first. In SPIR-V a direct assignment consists of two instructions, `OpLoad` to load a value from memory, in this case `y`, followed by `OpStore` to another variable, in this case `x`. While `OpLoad` always has to occur before `OpStore`, other instructions may be called before the actual execution of `OpStore`. The copy propagation pass needs to check if `y` does not get manipulated in any way before `OpStore` is called. If `y` was not manipulated it is indeed a direct assignment, the copy propagation pass can use the result from `OpLoad`, `y`, and replace all uses of `x` with it. The algorithm used during this work, replaces all uses of `x` but stops the replacement if one of the following conditions is met. If another `OpStore` is found which sets `x` to a new value or if an `OpFunctionCall` is detected, which uses `x` as a parameter. This is needed since `x` could eventually be modified in the called function. Another condition to break out of replacement is the start of a new SPIR-V block, indicated by `OpLabel`, here the algorithm also has to stop replacement to prevent any errors considering the scope of the variable. The copy propagation pass also considers calls to structs and arrays. These calls are using `OpAccessChain` instructions in SPIR-V. While the idea for replacing calls to `OpAccessChain` remains the same as with regular `OpStore` instructions, we need to check if the whole struct or array does not get modified to replace it safely. The copy propagation pass does not remove the actual `OpStore` or `OpAccessChain` instructions, instead it just replaces the used unique result identifiers. This means it is usually a good idea to run another dead code elimination pass when the copy propagation pass has finished. The dead code elimination pass will then detect the now unused `OpStore` and remove it along with the variable definition. Figure 4.8 shows the control flow of the implemented copy propagation pass and figure 4.9 provides an overview of the copy propagation pass class.

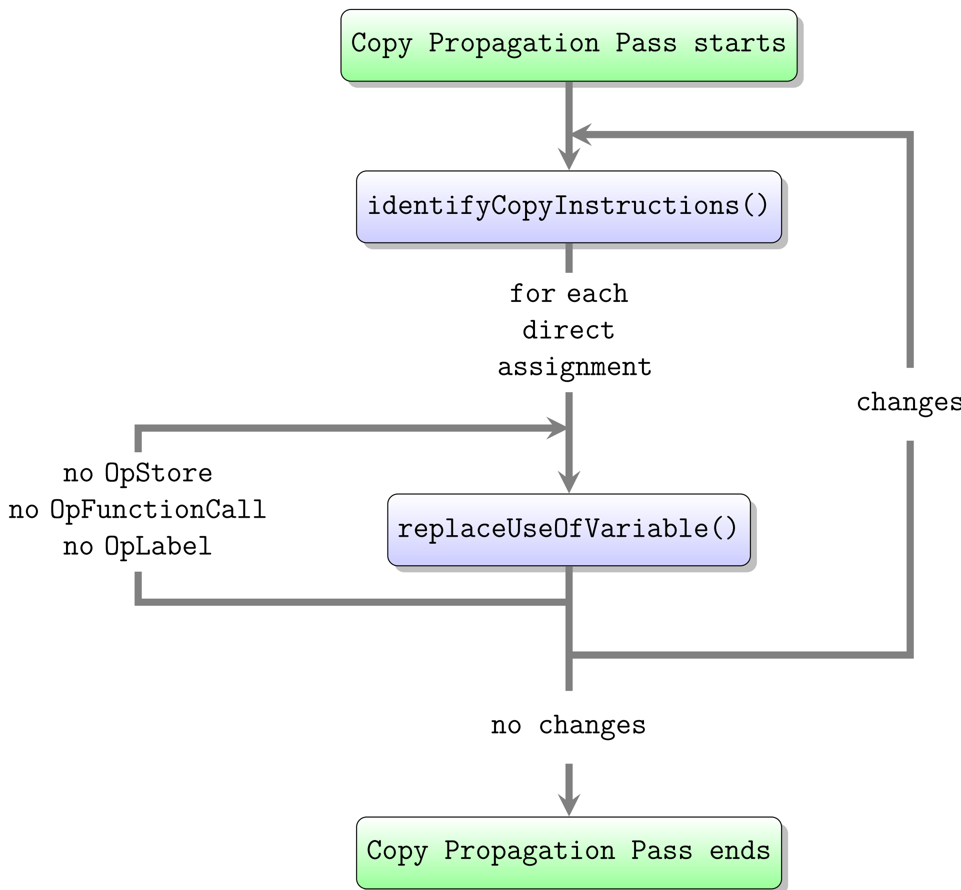


Figure 4.8: This figure shows the basic control flow of the copy propagation pass.

4.2.4 Remap Ids Pass

According to the SPIR-V specification, the unique identifiers of the instructions in the SPIR-V binary should be kept as low as possible. When instructions are removed from the binary with any optimization pass, it will most likely happen that not all potential unique identifiers are used. As an example, if with have a SPIR-V binary where the unique identifiers reach from 1 to 50 and we remove the instructions 44 and 45, it would make sense to remap the identifiers to the range of 1 to 48. This is also important after an inline function pass was executed. The inline function pass copies entire function bodies, hence it needs to acquire new unique identifiers for each copied instruction. As a result the identifiers can grow large quickly if a lot of functions are inlined. The remap ids pass makes sure there are no gaps between unique identifiers to guarantee the minimal possible range. The algorithm used to

Optimization Passes

CopyPropagationPass

private:
mModule : ModulePtr
mDefUseMgr : DefUseMgrPtr
mCopyInstructions : InstructionQueue

public:
Process(module : ModulePtr) : Status

private:
identifyCopyInstructions() : void
replaceVariables() : bool

Figure 4.9: Copy propagation pass class.

implement the remap ids pass is rather simple. It parses through all the instructions in the SPIR-V binary and remembers all used unique identifiers. As a next step, it creates a list of all unused identifiers from 1 to the highest unique identifier found. It then steps through all the used identifiers, checking if there is any unused identifier with a lower value. If there is a lower one, the used identifier is replaced with this lower value and the overridden identifier is added to the list of free identifiers. These steps are done for all used identifiers, resulting in the lowest possible range of unique identifiers. Figure 4.10 shows the control flow of the remap ids pass and figure 4.11 provides an overview of the remap ids pass class.

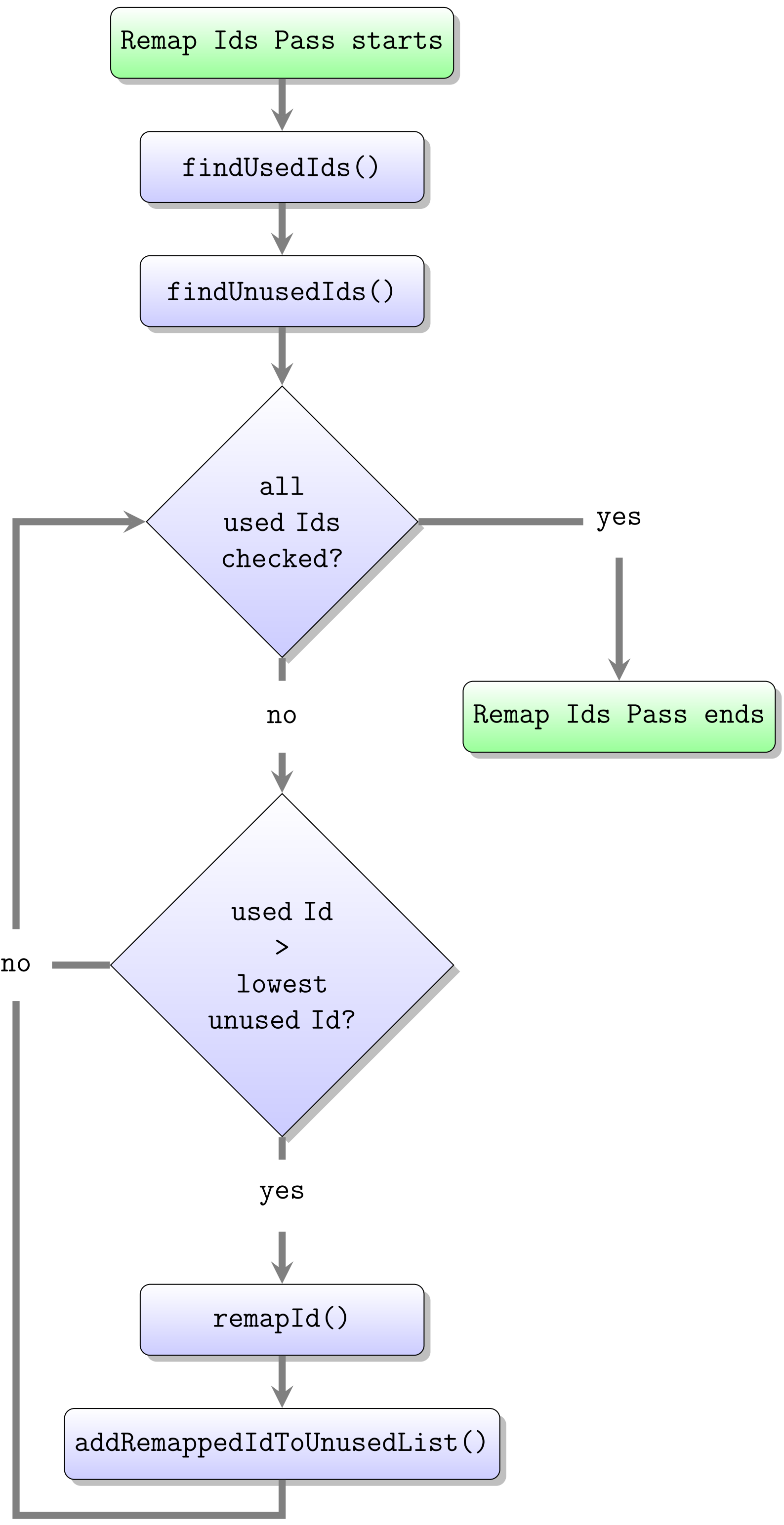


Figure 4.10: This figure shows the basic control flow of the remap ids pass.

Optimization Passes

RemapIdsPass

private:
mModule : ModulePtr
mDefUseMgr : DefUseMgrPtr

public:
Process(module : ModulePtr) : Status

private:
bool remapIds() : bool
copyInstructionAndOperands(inst : InstructionPtr,
oldToNewIds Map<uint, uint>) : InstructionPtr

Figure 4.11: Remap Ids pass class.

4.2.5 Helper Classes

As discussed in section 4.2.1 and section 4.2.2 some additional helper classes were needed while parsing SPIR-V instructions. The basic block info class was used to store information about parent and child relationships of blocks during the dead code elimination pass. The function info class was mainly used for the inline functions pass and allows to navigate through the function call graph. The pass utils class was used in all the optimization passes and provides methods which simplify accessing SPIR-V instructions and their operands.

4.2.5.1 Basic Block Info Class

Figure 4.12 displays an overview of the basic block info class. This class keeps track of the relationship between SPIR-V basic blocks. In SPIR-V a basic block always starts with an `OpLabel` instruction and ends with a branch instruction. The basic block info stores these instructions along with the merge instruction, if the branch instruction was a conditional branch. Furthermore the basic block info class holds information about the parent block, which is the block branching into the current basic block. All the child blocks the current block directly branches into are saved likewise. There can be multiple child blocks in case of `OpSwitch` instructions or

`OpBranchConditional` instructions, but there is only one parent block at all times. The basic block info class uses the `OpLabel` instruction of the parent or child to identify it. Since the dead code elimination pass needs to step through basic blocks and mark them dead or alive, the basic block info also keeps a `mIsDead` flag to keep track of this information.

Helper Classes

BasicBlockInfo

private:
mIsDead : bool
mNeedsCleanup : bool
mLabel : InstructionPtr
mParentBlock : InstructionPtr
mMergeInstruction : InstructionPtr
mBranchInstruction : InstructionPtr
mChildBlocks : InstructionVector

private:
getChildBlockById(blockId : uint) : InstructionPtr

public:
BasicBlockInfo(label : InstructionPtr, parentBlock : InstructionPtr)
getBlock() : InstructionPtr
setParentBlock(parentBlock : InstructionPtr) : void
getParentBlock() : InstructionPtr
getParentBlockId() : uint

setMergeInstruction(mergeInstruction : InstructionPtr) : void
getMergeInstruction() : InstructionPtr
hasMergeInstruction() : bool

setBranchInstruction(branchInstruction : InstructionPtr) : void
getBranchInstruction() : InstructionPtr
hasBranchInstruction() : bool

addChildBlock(childBlock : InstructionPtr) : void
removeChildBlock(childBlock : InstructionPtr) : void
removeChildBlockById(blockId : uint) : void
getNumberOfChildBlocks() : uint
getChildBlockByIndex(index : uint) : InstructionPtr
hasChildBlock(childBlock : InstructionPtr) : bool
hasChildBlock(blockId : uint) : bool

markBlockDead() : void
isBlockDead() : bool
markBlockAlive() : void
isBlockAlive() : bool
markForCleanup() : void
needsCleanup() : bool

Figure 4.12: The basic block info class.

4.2.5.2 Function Info Class

As the name suggests the function info class provides additional information about the functions defined in the SPIR-V binary. It holds a list of all the functions that call the current function and also all the functions called by the current function itself. The inline functions pass makes heavy use of this function info class, when deciding which function is the next one to inline. The function info class also provides a `mIsInlined` flag, so functions can be marked as inlined by the inline functions pass. Figure 4.13 provides an overview of the function info class.

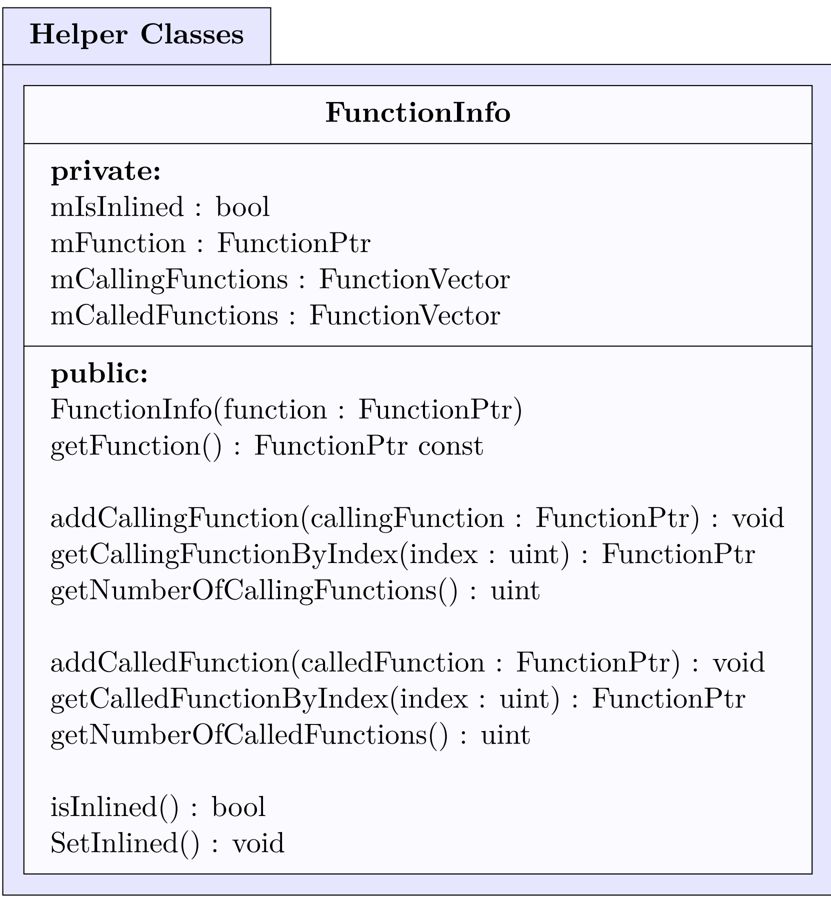


Figure 4.13: The function info class.

4.2.5.3 Pass Utils Class

As opposed to the basic block info class and the function info class, the pass utils class is not designed to support one specific optimization pass. Instead the pass

utils class provides various static methods to simplify working with instructions and their operands. Especially accessing and comparing operands of instructions becomes much more readable when using this class. The pass utils class also provides two methods for printing debug information to the debug console, `DebugPrintInstruction` and `DebugPrintModule`. `DebugPrintInstruction` prints the written name of the instruction along with all its operands, while `DebugPrintModule` steps through the entire SPIR-V module and prints out every instruction found. Figure 4.14 provides an overview of the pass utils class.

Helper Classes

PassUtils

public:
DebugPrintInstruction(text : string, inst : InstructionRef) : static void
DebugPrintModule(module : ModulePtr) : static void

GetOperandId(inst : InstructionRef, num : uint) : static uint
GetFirstInOperandId(inst : InstructionRef) : static uint
GetSecondInOperandId(inst : InstructionRef) : static uint
GetThirdInOperandId(inst : InstructionRef) : static uint
GetInOperandId(inst : InstructionRef, num : uint) : static uint

CompareFirstInOperandIds(left_inst : InstructionRef,
right_inst : InstructionRef) : static bool
CompareInOperandIds(left_inst : InstructionRef, left_num : uint,
right_inst : InstructionRef, right_num : uint) : static bool

Figure 4.14: The pass utils class.

4.3 Watchpoints

Debugging shader code is hard task to do, independent of the shading language used. When a programmer writes a new fragment shader and he ends up having an error somewhere in the shader code, the output of the shader will most likely be some random colors or there is no output at all. The usual steps the programmer takes to find the error are going through every line of shader code and set the shader output color to intermediate values of the shader to check the intermediate results. This is time consuming and inconvenient. As the Murl Engine also provides a graphical interface for developers, a possibility to automate this debugging procedure of shader code would be of great use. For this reason watchpoints for GLSL fragment shaders were introduced during this work. The idea is to let programmers define a line and a column in their shader via the graphical interface of the Murl Engine, the intermediate value of the variable at the specific line and column is stored and displayed as debug output somewhere within the graphical interface. The SPIR-V Wrapper application implements the first steps of this approach. As seen in table 4.2 the interface of the SPIR-V Wrapper allows users to set the line and column of a watchpoint.

Example 11 GLSL fragment shader before watchpoint injection.

```
1: out vec4 gl_FragColor;
2: void main()
3: {
4:     vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
5:     vec4 green = vec4(0.0, 1.0, 0.0, 1.0);
6:     gl_FragColor = red + green;
7: }
```

SPIR-V Wrapper will inject the shader with a global variable defined as `out vec4 watchPoint`. While compiling the shader to the abstract syntax tree (AST) using the GLSL Reference Compiler, SPIR-V Wrapper detects the variable located at the line and column set by the programmer and adds an assignment from this variable to the newly created `vec4 watchPoint`. SPIR-V Wrapper can only watch float, vec2, vec3 and vec4 type variables and will return an error if the variable is of any other type. If the watched variable is of type float, SPIR-V Wrapper uses swizzles to save the value, resulting in an assignment like `watchPoint.x = watchedFloatVariable`. The same

applies for vec2 and vec3 type variables. Examples 11 and 12 show a GLSL fragment shader before and after watchpoint injection. The `watchPoint` variable will always be initialized with `vec4(1.0)`. To implement the watchpoint system, the grammer of the GLSL Reference compiler was changed in a way, that after every initialization of a variable, every assignment to a variable and after every post increment or post decrement statement there could theoretically be a watchpoint inserted. This work does not further process the the `watchPoint` variable. As already mentioned the graphical interface of the Murl Engine is meant to read the `watchPoint` variable and visually present its values in the future.

Example 12 Example 11 with watchpoint inserted at line 5 and column 5.

```
1: out vec4 watchPoint;
2: out vec4 gl_FragColor;
3: void main()
4: {
5:     vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
6:     vec4 green = vec4(0.0, 1.0, 0.0, 1.0);
7:     watchPoint = vec4(1.0);
8:     watchPoint = green;
9:     gl_FragColor = red + green;
10: }
```

Chapter 5

Results

This chapter describes the test environment used to measure the performance of SPIR-V shaders and brings SPIR-V shaders with and without optimizations face to face. Section 5.1 briefly explains how the test environment was set up and which tools were used, while section 5.2 showcases the shaders used for testing. Subsequently sections 5.3 and 5.4 evaluate the results of the optimizations implemented during this work.

5.1 Test Environment

As stated in chapter 4 the SPIR-V Wrapper, created during this work, does multiple conversion steps. First the a shader is converted to SPIR-V, then it gets optimized and after that it will be converted back to GLSL or HLSL again. When trying to measure the performance of a shader this setup makes it difficult to exactly pin down where performance was gained or lost. For instance, it could be possible that the back-end conversion done by SPIR-V Cross is not optimal and thus some previously done optimizations lose their effect. Following this thought, it makes more sense to compare the SPIR-V binaries before and after optimization directly. As SPIR-V is the native shading language for the Vulkan API, a Vulkan rendering pipeline was needed. For this work a program called Vulkan Shader Evaluator was created from scratch. The Vulkan Shader Evaluator is using VulkanSDK 1.0.68.0 [27] as well as GLFW 3.2.1 [10] for window creation. All testing was done on Windows 10 running a Nvidia Geforce GTX 760 with the newest drivers available. The Vulkan Shader

Evaluator renders all output to a window of size 1024x768. While this setup enables SPIR-V shaders to be compared visually it still does not allow in depth comparison of the shaders performance. To actually measure the frame rate (frames per second) of the rendered output, another tool called PresentMon was needed. PresentMon is small command line tool which tracks the performance of a rendering process by measuring the micro seconds between presenting images for a specific amount of time. For this evaluation PresentMon 1.3.0 was used [9].

5.2 Test Cases

To compare the SPIR-V binaries a set of 20 shaders was gathered for testing, further referenced as Ex01-Ex20. All of these shaders are publicly available on the Shadertoy website [15]. The full list of shaders can be found in the appendix A of this thesis. As the shaders available on Shadertoy are mostly fragment shaders, all test cases are using the same vertex shader which is shown in example 13. The Vulkan Shader Evaluator creates a window sized rectangle and passes its vertex data to the vertex shader. Afterwards the vertex shader just transforms coordinates from world space to screen space and passes all needed data on to the fragment shader. The fragment shaders then procedurally generates outputs as shown in 5.1. The performance of each shader was measured over the duration of 120 seconds starting from the first frame rendered. Afterwards the average frame rate was calculated. To have a reference value all shaders were first rendered without any optimization passes applied. As a second step, the shaders were optimized by the SPIR-V Wrapper using the optimization passes as shown in table 5.1. The optimizations were applied in this exact order and the measurement was repeated. As a second parameter the byte size of the SPIR-V binary code was examined before and after optimization.

Pass Number	Optimization Pass Type
1	`OptPassTypeEliminateDeadCode`
2	`OptPassTypeUnifyConstant`
3	`OptPassTypeStripDebugInfo`
4	`OptPassTypeFoldSpecConstantOpAndComposite`
5	`OptPassTypeEliminateDeadConstant`
6	`OptPassTypeEliminateDeadCode`
7	`OptPassTypeCopyPropagation`
8	`OptPassTypeEliminateDeadCode`
9	`OptPassTypeRemapIds`

Table 5.1: This table shows the order of the optimization passes applied for evaluation.

Example 13 The vertex shader used by the Vulkan Shader Evaluator.

```
1: #version 450
2: #extension GL_ARB_separate_shader_objects : enable

3: layout(binding = 0) uniform UniformBufferObject {
4:     mat4 model;
5:     mat4 view;
6:     mat4 proj;
7: } ubo;

8: layout(binding = 2) uniform ShaderToyGlobals {
9:     vec3 resolution;
10:    float time;
11: } shaderToyGlobals;

12: layout(location = 0) in vec2 inPosition;
13: layout(location = 1) in vec3 inColor;
14: layout(location = 2) in vec2 inTexCoord;

15: layout(location = 0) out vec3 fragColor;
16: layout(location = 1) out vec2 fragTexCoord;

17: out gl_PerVertex {
18:     vec4 gl_Position;
19: };

20: void main() {
21:     gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
22:     fragColor = inColor;
23:     fragTexCoord.x = inTexCoord.x * shaderToyGlobals.resolution.x;
24:     fragTexCoord.y = inTexCoord.y * shaderToyGlobals.resolution.y;
25: }
```

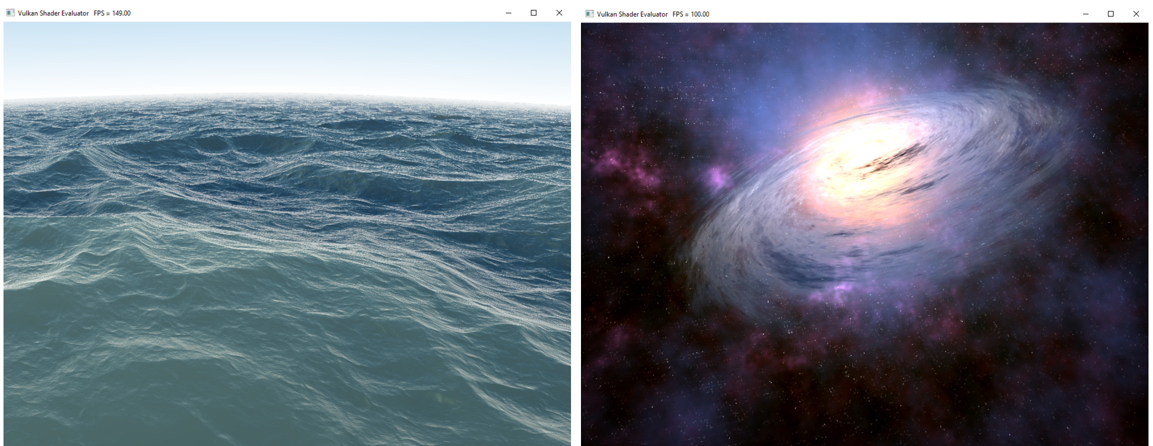


Figure 5.1: Output of test shaders Ex01(left) and Ex19(right) used for evaluation.

5.3 Evaluation of Frame Rates

Figure 5.2, 5.3 and 5.4 are visualizing the results of the measurements described in the previous sections. While the blue bars represent the frame rate of the non-optimized SPIR-V binary shader, the green bars are showing the optimized versions. It is easy to see that the differences in in terms of frame rate are not very distinct, however the optimized versions gain a slight edge over their non-optimized counterparts. The biggest differential appears at Ex18, with about 0.75 frames per second which visually does not make a difference. This result however is not very surprising. Since state of the art compilers are already very good at optimizing code the results are expected to be very similar. On an older compiler the results would probably be a very different. Considering that the Murl Engine has its focus on mobile applications, one has to be aware that the amount of different compilers in mobile devices is enormous. Many devices rely on poor compilers, hence the SPIR-V optimizations are helping to reduce the dependency on the implementation of those compilers.

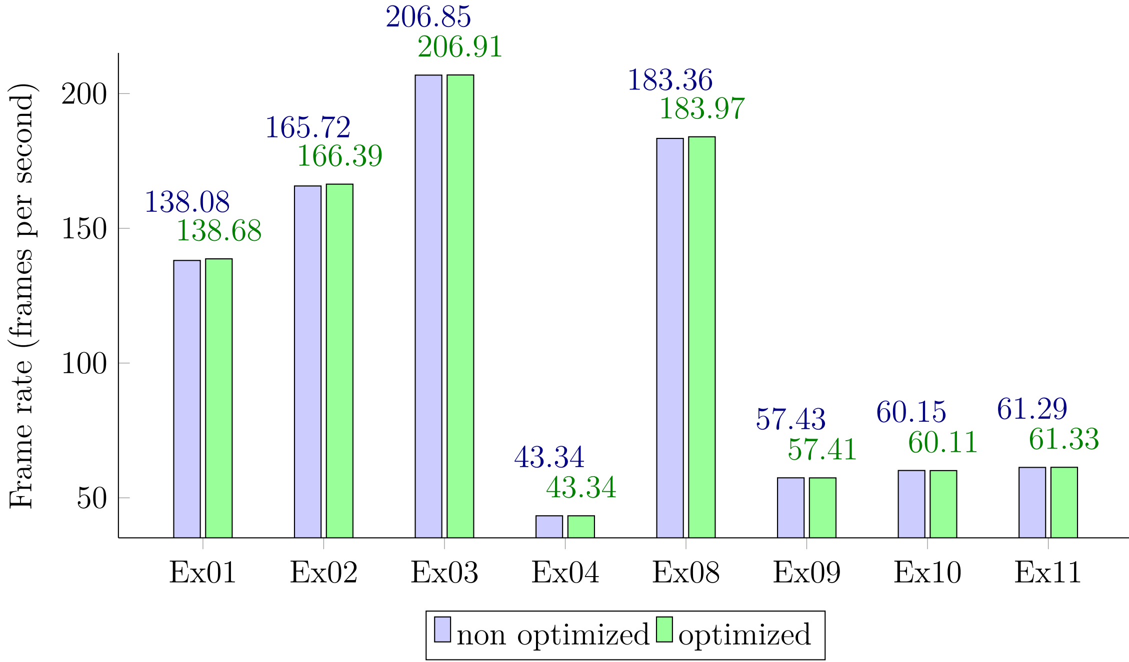


Figure 5.2: Comparison of frame rates of non-optimized and optimized SPIR-V shaders for test shaders Ex01, Ex02, Ex03, Ex04, Ex08, Ex09, Ex10 and Ex11.

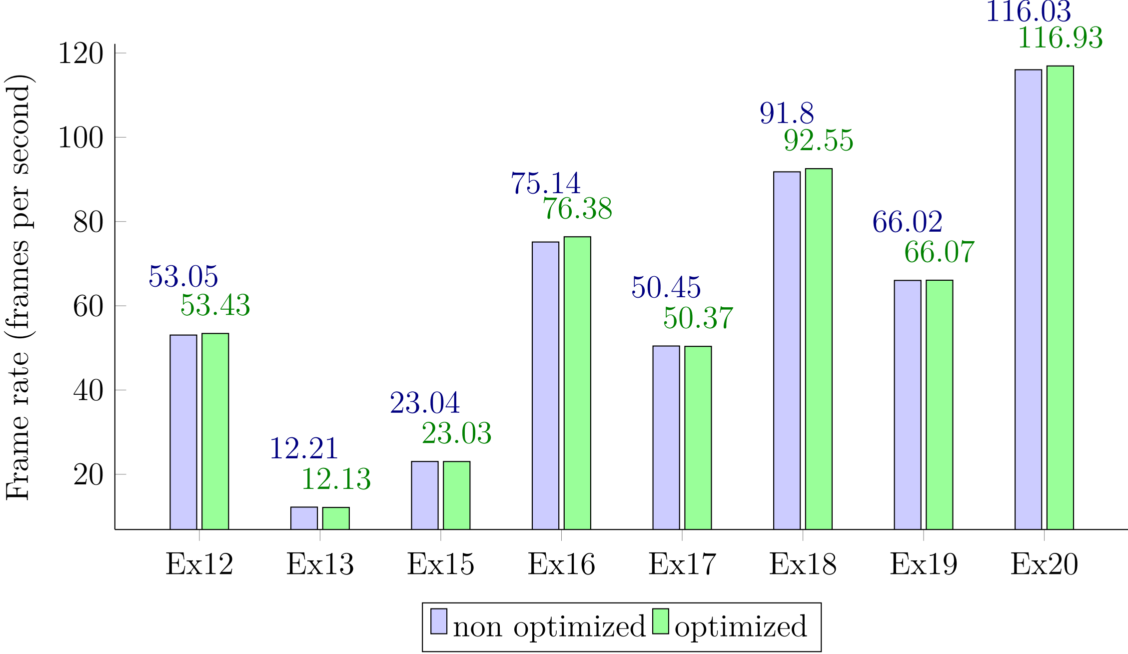


Figure 5.3: Comparison of frame rates of non-optimized and optimized SPIR-V shaders for test shaders Ex12, Ex13, Ex15, Ex16, Ex17, Ex18, Ex19 and Ex20.

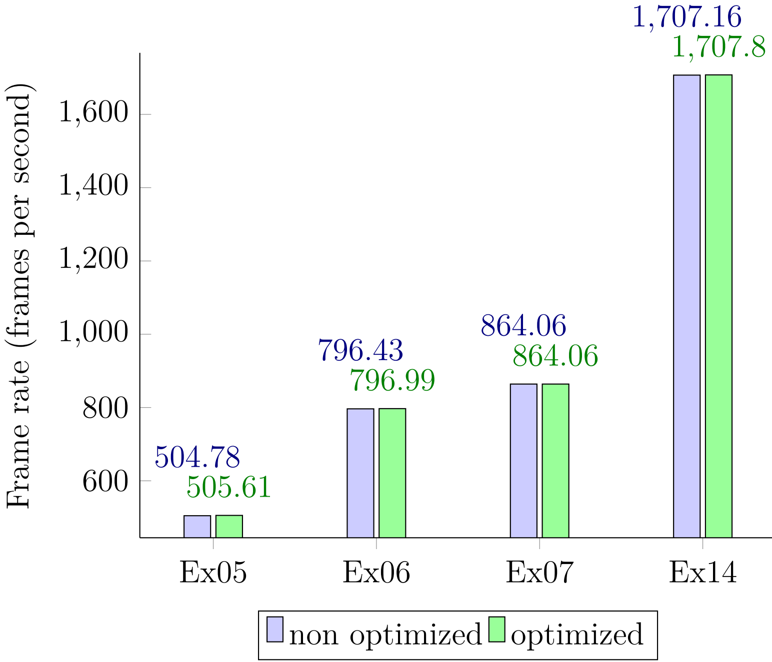


Figure 5.4: Comparison of frame rates of non-optimized and optimized SPIR-V shaders for test shaders Ex05, Ex06, Ex07 and Ex14.

5.4 Evaluation of Binary Sizes

Figure 5.5, 5.6 and 5.7 are comparing the byte size of the binary SPIR-V modules. Since the optimization passes implemented during this work, remove many unused instructions the byte size of the optimized binary is expected to be lower. The blue bars in the figure are representing the original byte size, while the orange bars are showing the optimized version. It is evident from the charts, that byte size of the optimized SPIR-V is reduced significantly. For the test shaders used during this work the minimal difference in size occurs in Ex07, being 4.18%, and the maximal difference appears in Ex12, with a 20.88% smaller than the original size. On average using SPIR-V Wrapper leads to a reduction in file size of 11.1%. This can further lead to increased performance when many different shaders need to be loaded and unloaded in real time. While not being a problem on desktop systems, mobile developers are usually struggling to keep the size of their applications as low as possible, consequently every byte counts. This is also the reason the inline functions pass was not used during evaluation. As examples showed, blindly inline expanding

all functions lead to code bloating, resulting in huge binary byte sizes and overall worse performance. However, the inline function pass is still of great use to the Murl Engine, since it reduces a shader to just a single main function. As the Murl Engine internally splits variable definitions and function implementations of shaders the optimization makes this separation easier.

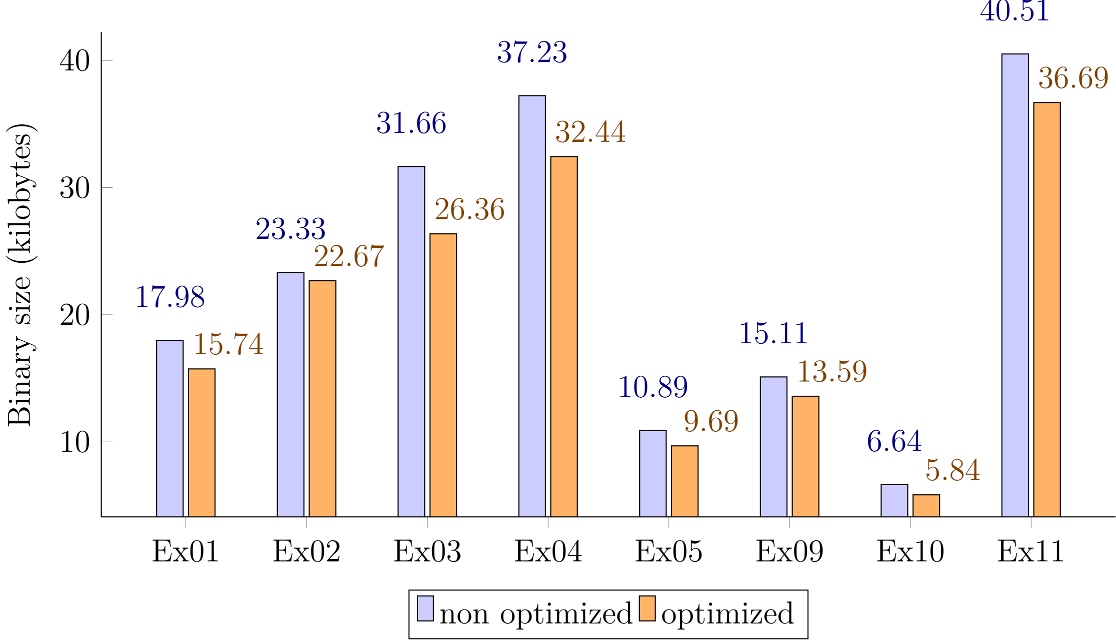


Figure 5.5: Comparison of binary sizes of non-optimized and optimized SPIR-V shaders for test shaders Ex01, Ex02, Ex03, Ex04, Ex05, Ex09, Ex10 and Ex11.

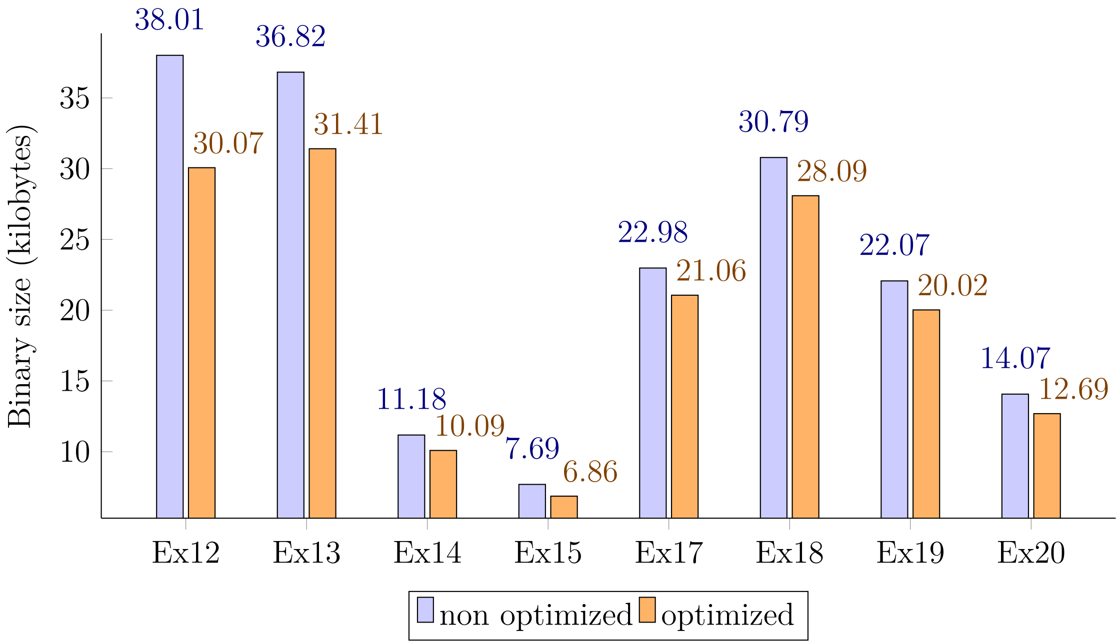


Figure 5.6: Comparison of binary sizes of non-optimized and optimized SPIR-V shaders for test shaders Ex12, Ex13, Ex14, Ex15, Ex17, Ex18, Ex19 and Ex20.

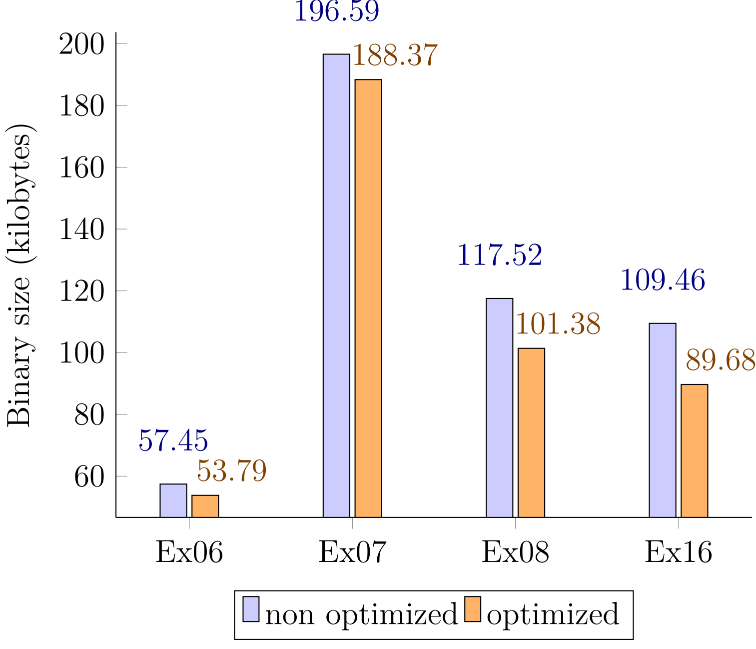


Figure 5.7: Comparison of binary sizes of non-optimized and optimized SPIR-V shaders for test shaders Ex06, Ex07, Ex08 and Ex16.

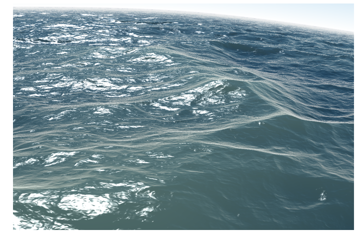
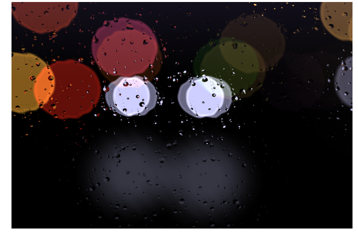
Chapter 6

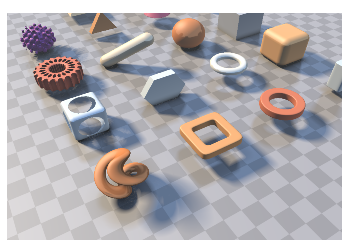
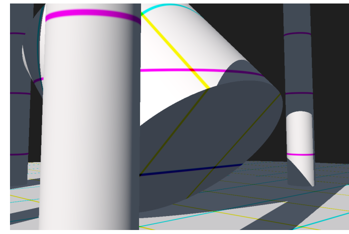
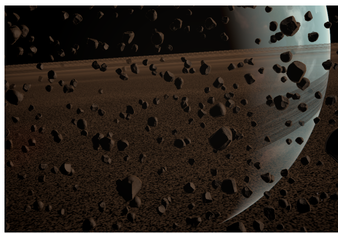
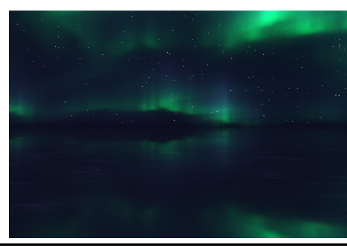
Conclusion

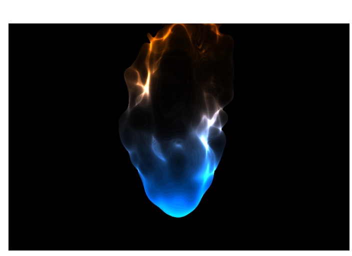
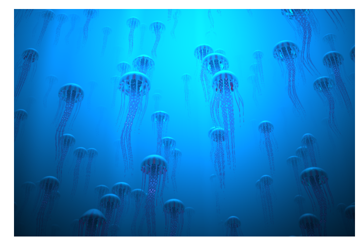
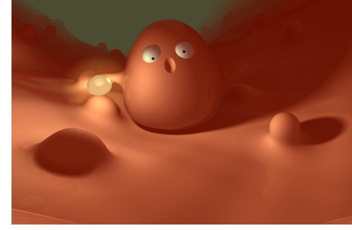
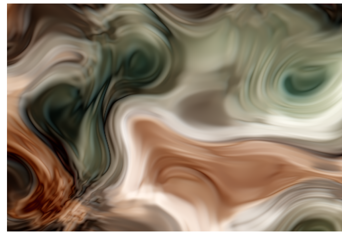
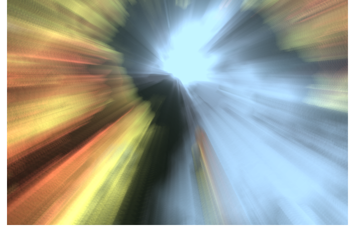
This work has shown the development of a third party enhancement for the cross-platform multimedia framework Murl Engine. This thesis shows how the Murl Engine can be more flexible in terms of supported shader languages, by using the native Vulkan shader language SPIR-V as an intermediate representation. Currently available tools for working with SPIR-V shaders were examined and three of them where combined to perform translations between various shader languages. A setup was introduced using the GLSL Reference Compiler as a front-end, the SPIR-V Tools project for optimization and validation and the SPIRV-Cross framework for back-end conversion. It was found the SPIR-V Tools project was lacking optimization options in comparison to the GLSL Optimizer used by the Murl Engine. Therefore multiple optimization passes, including dead code elimination, inline expansion, copy propagation and id remapping were added to the SPIR-V Tools framework.These optimization passes were evaluated using a Vulkan test environment, running the SPIR-V shaders directly and comparing them to their non-optimized counterparts. As expected the performance gain in terms of frame rates was just minor. However, dependency on the implementation of the used compilers was minimized due to offline optimization. The evaluation also pointed out that the byte size of the binary SPIR-V modules was significantly reduced by applying offline optimization. After optimizing 20 different SPIR-V fragment shaders, the reduction in byte size averaged at 11.1% which can be considered a great success.

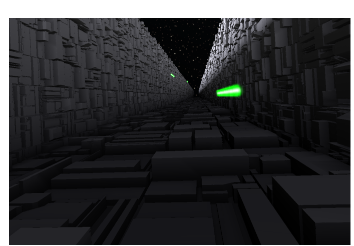
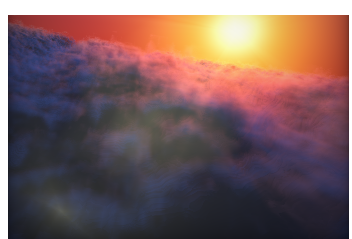
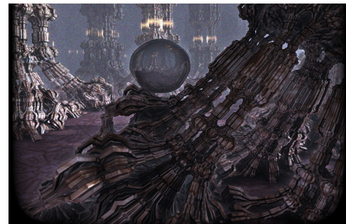
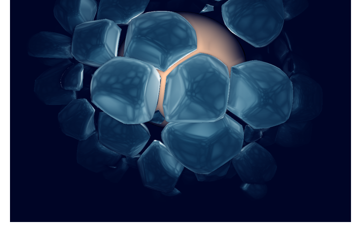
Appendix A

List of Test Shaders

Output	Description
	**Example 01 (Ex01):** Seascape by Alexander Alekseev aka TDM `https://www.shadertoy.com/view/Ms2SD1`
	Example 02 (Ex02): Kirby Jump by fizzer `https://www.shadertoy.com/view/lt2fD3`
	**Example 03 (Ex03):** The Drive Home by Martijn Steinrucken aka Big-Wings `https://www.shadertoy.com/view/MdfBRX`

Output	Description
	**Example 04 (Ex04):** Raymarching - Primitives by Inigo Quilez aka iq `https://www.shadertoy.com/view/Xds3zN`
	**Example 05 (Ex05):** Simple Cylinder Ray Tracer by Hazel Quantock aka TekF `https://www.shadertoy.com/view/Md3cWj`
	Example 06 (Ex06): Wolfenstein 3D by Reinder Nijhoff aka reinder `https://www.shadertoy.com/view/4sfGWX`
	Example 07 (Ex07): [SIG15] Mario World 1-1 by Krzysztof Narkowicz aka knarkowicz `https://www.shadertoy.com/view/XtlSD7`
	**Example 08 (Ex08):** Planet Shadertoy by Reinder Nijhoff aka reinder `https://www.shadertoy.com/view/4tjGRh`
	**Example 09 (Ex09):** Auroras by nimitz `https://www.shadertoy.com/view/XtGGRt`

Output	Description
	**Example 10 (Ex10):** Flame by XT95 `https://www.shadertoy.com/view/MdX3zr`
	Example 11 (Ex11): Mike by Inigo Quilez aka iq `https://www.shadertoy.com/view/MsXGWr`
	**Example 12 (Ex12):** Luminescence by Martijn Steinrucken aka BigWings `https://www.shadertoy.com/view/4sXBRn`
	**Example 13 (Ex13):** Bouncy boi on the run by stellabialek `https://www.shadertoy.com/view/XscyRs`
	**Example 14 (Ex14):** Warping - procedural 2 by Inigo Quilez aka iq `https://www.shadertoy.com/view/lsl3RH`
	**Example 15 (Ex15):** Radial Blur 2k18 by Przemyslaw Zaworski aka PrzemyslawZaworski `https://www.shadertoy.com/view/MtjfRd`

Output	Description
	**Example 16 (Ex16):** Simple Greeble - Split4 by Jerome Liard aka blackjero `https://www.shadertoy.com/view/4tXcRl`
	**Example 17 (Ex17):** Cloud Ten by nimitz `https://www.shadertoy.com/view/XtS3DD`
	**Example 18 (Ex18):** Generators by Kali `https://www.shadertoy.com/view/Xtf3Rn`
	Example 19 (Ex19): Pegasus Galaxy by Frank Hugenroth by frankenburgh `https://www.shadertoy.com/view/lty3Rt`
	**Example 20 (Ex20):** Ray Marching Experiment no 76 by Stephane Cuillerdier aka aiekick `https://www.shadertoy.com/view/4stcRr`

Bibliography

[1] A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, first edn., 1999. ISBN 1558606181. Cited on page 5.

[2] Apple Inc. Metal 2 Reference. `https://developer.apple.com/metal/`, 2017. Accessed: 2017-05-16. Cited on page 11.

[3] G. Bilardi and K. Pingali. The static single assignment form and its computation. Tech. rep., Università di Padova, Cornell University, 1999. Cited on page 12.

[4] J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in isabelle/hol. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pp. 200–209, 2005. doi:10.1109/SEFM.2005.20. Cited on page 18.

[5] D. Blythe. The direct3d 10 system. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pp. 724–734. ACM, New York, NY, USA, 2006. ISBN 1-59593-364-6. doi:10.1145/1179352.1141947. Cited on page 6.

[6] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pp. 246–257. ACM, New York, NY, USA, 1989. ISBN 0-89791-306-X. doi:10.1145/73141.74840. Cited on page 19.

[7] R. L. Cook. Shade trees. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pp. 223–231.

ACM, New York, NY, USA, 1984. ISBN 0-89791-138-5. doi:10.1145/800031.808602. Cited on page 5.

[8] R. L. Cook, L. Carpenter, and E. Catmull. The reyes image rendering architecture. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pp. 95–102. ACM, New York, NY, USA, 1987. ISBN 0-89791-227-6. doi:10.1145/37401.37414. Cited on page 5.

[9] GameTechDev. PresentMon Project. `https://github.com/GameTechDev/PresentMon`, 2018. Accessed: 2018-01-20. Cited on page 55.

[10] GLFW Project. GLFW. `http://www.glfw.org/`, 2018. Accessed: 2018-01-20. Cited on page 54.

[11] Google LLC. ShaderC Project. `https://github.com/google/shaderc`, 2017. Accessed: 2017-05-13. Cited on page 28.

[12] GraphViz. Graph Visualization Software. `https://www.graphviz.org/documentation/`, 2017. Accessed: 2017-05-13. Cited on page 26.

[13] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pp. 289–298. ACM, New York, NY, USA, 1990. ISBN 0-89791-344-2. doi:10.1145/97879.97911. Cited on page 5.

[14] J. Hunz. The possibilities of compute shaders - an analysis, 2013. Cited on page 10.

[15] P. J. Ingo Quilez. Shadertoy. `https://www.shadertoy.com`, 2018. Accessed: 2018-01-20. Cited on page 55.

[16] Khronos Group. ESSL Reference. `https://www.khronos.org/registry/OpenGL/index_es.php`, 2017. Accessed: 2017-07-23. Cited on page 11.

[17] Khronos Group. GLSL Reference Compiler. `https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/`, 2017. Accessed: 2017-05-13. Cited on page 23.

[18] Khronos Group. GLSL Reference Compiler GitHub. `https://github.com/KhronosGroup/glslang`, 2017. Accessed: 2017-05-13. Cited on page 23.

[19] Khronos Group. GLSL Specification. `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf`, 2017. Accessed: 2017-07-23. Cited on pages 3 and 11.

[20] Khronos Group. SPIR-V Cross Project. `https://github.com/KhronosGroup/SPIRV-Cross`, 2017. Accessed: 2017-05-13. Cited on page 25.

[21] Khronos Group. SPIR-V Specification. `https://www.khronos.org/registry/spir-v/`, 2017. Accessed: 2017-05-16. Cited on pages 4 and 12.

[22] Khronos Group. SPIR-V Tools Project. `https://github.com/KhronosGroup/SPIRV-Tools`, 2017. Accessed: 2017-05-13. Cited on page 26.

[23] KTX Software. Kha Framework. `http://kha.tech/`, 2017. Accessed: 2017-05-13. Cited on page 26.

[24] KTX Software. Krafix Project. `https://github.com/KTXSoftware/krafix`, 2017. Accessed: 2017-05-13. Cited on page 28.

[25] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pp. 149–158. ACM, New York, NY, USA, 2001. ISBN 1-58113-374-X. doi:10.1145/383259.383274. Cited on page 5.

[26] LLVM Developer Group. LLVM Language Reference. `https://llvm.org/docs/LangRef.html`, 2017. Accessed: 2017-05-13. Cited on page 12.

[27] Lunar G. Vulkan SDK. `https://www.lunarg.com/vulkan-sdk/`, 2018. Accessed: 2018-01-20. Cited on page 54.

[28] W. R. Mark, R. Steven, G. Kurt, A. Mark, and J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, vol. 22, pp. 896–907, 2003. Cited on page 6.

[29] W. mei W. Hwu and P. P. Chang. Inline function expansion for compiling c programs. *ACM SIGPLAN Notices*, 1989. Cited on page 40.

[30] Microsoft Corporation. Reference for HLSL. `https://msdn.microsoft.com/en-us/library/windows/desktop/bb509638(v=vs.85).aspx`, 2017. Accessed: 2018-01-12. Cited on pages 3, 6, and 11.

[31] H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edn., 2007. ISBN 9780321545428. Cited on pages 9 and 10.

[32] M. Nießner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer. Real-time rendering techniques with hardware tessellation. *Computer Graphics Forum*, 2015. Cited on page 8.

[33] D. Novillo. A propagation engine for gcc. In *In Proceedings of the 2005 GCC Developers Summit*, 2005. Cited on pages 21 and 22.

[34] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pp. 425–432. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 1-58113-208-5. doi:10.1145/344779.344976. Cited on page 5.

[35] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. ISBN 0321335597. Cited on page 7.

[36] A. Pranckevičius. SMOL-V Project. `https://github.com/aras-p/smol-v`, 2016. Accessed: 2017-05-16. Cited on page 29.

[37] A. Pranckevičius. SPIR-V Compression. `http://aras-p.info/blog/2016/09/01/SPIR-V-Compression/`, 2016. Accessed: 2017-05-16. Cited on page 29.

[38] A. Pranckevičius. GLSL Optimizer. `http://aras-p.info/blog/2010/09/29/glsl-optimizer/`, 2017. Accessed: 2017-05-16. Cited on page 3.

[39] A. Pranckevičius. GLSL Optimizer GitHub. `https://github.com/aras-p/glsl-optimizer`, 2017. Accessed: 2017-05-16. Cited on page 3.

[40] R. J. Rost, B. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, third edn., 2009. ISBN 0321637631, 9780321637635. Cited on pages 6 and 11.

[41] D. Salomon. *Variable-length Codes for Data Compression*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 1846289580. Cited on page 29.

[42] Sassa Laboratory, Graduate School of Information Science and Engineering. Optimization in static single assignment form, external specification. Tech. rep., Tokyo Institute of Technology, 2007. Cited on pages 18 and 21.

[43] Spraylight GmbH. Murl Engine. `http://murlengine.com/`, 2017. Accessed: 2017-05-16. Cited on page 1.

[44] Spraylight GmbH. Spraylight GmbH. `http://spraylight.at/`, 2017. Accessed: 2017-05-16. Cited on page 1.

[45] S. Upstill. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0201508680. Cited on page 5.

[46] D. Wolff. *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, 2011. ISBN 1849514763, 9781849514767. Cited on page 6.

[47] L. D. Yunming Zhang, Rebecca Smith. Implementation of the Mark-Sweep Dead Code Elimination Algorithm in LLVM. Tech. rep., 2013. Cited on page 36.

[48] J. Zink, M. Pettineo, and J. Hoxley. *Practical Rendering and Computation with Direct3D 11*. A. K. Peters, Ltd., Natick, MA, USA, first edn., 2011. ISBN 1568817207, 9781568817200. Cited on page 6.