HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Simulating the GPU-based shaders in the graphics pipeline on a CPU-based language to allow code inspection at runtime

Masterthesis

at the University of applied science Ravensburg-Weingarten

by

Matthias Mettenleiter

September 2019

Student ID	29015
Supervisor	Daniel Scherzer
Secondary supervisor	Sebastian Mauser

Author's declaration

Hereby I solemnly declare:

1. that this Masterthesis, titled *Simulating the GPU-based shaders in the graphics pipeline on a CPU-based language to allow code inspection at runtime* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Masterthesis has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Masterthesis in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Weingarten, September 2019

Matthias Mettenleiter

Abstract

An abstract is a brief summary of a research article, thesis, review, conference proceeding or any in-depth analysis of a particular subject or discipline, and is often used to help the reader quickly ascertain the paper's purpose. When used, an abstract always appears at the beginning of a manuscript, acting as the point-of-entry for any given scientific paper or patent application. Abstracting and indexing services for various academic disciplines are aimed at compiling a body of literature for that particular subject.

The terms précis or synopsis are used in some publications to refer to the same thing that other publications might call an "abstract". In "management" reports, an executive summary usually contains more information (and often more sensitive information) than the abstract does.

Quelle: `http://en.wikipedia.org/wiki/Abstract_(summary)`

Contents

1 Introduction

Explanation of debugging "Debugging is the process of locating and removing faults in computer programs" according to [Collins 2014] . The steps that are part of the debugging process are reproducing the problem, identifying the source of the problem and fixing the problem. All of these steps can be done manually but there are ways to improve and accelerate this process.

To find problems there is the option of writing automated tests, inserting debug outputs on the console into the source code or writing states into log files. This enables the programmer to find anomalies before, while and after running the program.

When a way is found to reproduce the problem, to find the source of it the manual way is to increase the amount of debug outputs around the problematic part of the code and confine the point in the code at which the error occurs.

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" according to [Kernighan 1982] . As this states, debugging is a quite exhausting and time consuming task. For that reason for most programming languages there are tools to aid the programmer to narrow down the source of the bug with following methods:

- Enabling the user to set breakpoints at which the program pauses and he can inspect the values of the variables directly within the code. By continuing the program to move to the next breakpoint or by going forward through the code step by step the point where the error occurs can be found. It is also possible to be able to add conditions to the breakpoints describing a state that has to be fulfilled for the debugger to pause. [Undo 2019]
- Have the code throw an exception when unwanted behavior occurs and stop at this exception. By saving a stack of the calls which occurred before the exception was thrown or dumping the buffer, the programmer can retrace where the error may be found. [Jetbrains 2019]

- Reverse debugging records all program activities and thereby it is possible to move backwards in addition to forward stepping from a set breakpoint and see the changes in the variables and the calls in the code leading to the problem. [Undo 2019]

When the source of the problem is found the final step of fixing the problem is to correct the code.

Structure of the graphics pipeline The graphics pipeline also known as rendering pipeline differentiates from framework to framework but the basic structure is identical. [Khronos 2019a] [Microsoft 2018]

In the first stage the vertices are specified. The vertices are set up in an ordered list. By setting a primitive type it is determined how these vertices define the boundaries of primitives. Attributes are linked to the vertices adding data to them.

The second stage always is the stage where the vertex shader is executed for each defined vertex with its data. Each vertex is mapped to a specific output vertex with the output data of the vertex shader per vertex.

After the vertex stage multiple otional stages can occur. Examples for these are the Tesselation stage or the Geometry stage where additional shaders are executed to implement additional changes to the vertices or even remove or add vertices.

In the following stage which can also be regarded as multiple stages there are multiple optimization steps, the primitive assembly and the rasterization. The perspective divide and the viewport transformation are calculated. Through clipping, which is optional, primitives that overlap the edge of the viewing volume are split into multiple primitives and primitives outside the viewing volume are removed. Primitives are assembled according to the primitive type, so a list of primitives is constructed out of the list of vertices where each primitive is constructed out of multiple vertices. Face culling can occur as additional optimization removing primitives that face certain directions in the window space. The primitives are then rasterized into fragments within the raster of the render result. The data values linked to the vertices are atomatically interpolated within the primitive and added to the corresponding fragments.

Within the following fragment stage the fragment shader will be executed for each single fragment generating the outputs. There is always a depth buffer and a stencil buffer as an output. Usually a color output is also defined but it is possible to execute the pipeline with a default fragment shader where only depth and stencil buffers wil be set.

After the fragment stage the output of the render process is calculated. For each sample there are multiple otional filters that can be activated to change which fragments are used within this stage. Examples for these filters are the scissor test where a fragment is ignored when it's pixel lies outside of the rectangle of the screen or the depth test where the fragments order is changed when it's depth fulfils specific user defined conditions compared to another fragment's depth within the same sample. After all active filters were executed the color blending happens where the color between multiple fragments within the same sample is calculated together with a specific blending operation. The final data is written to the framebuffer.

The two most known uses of the graphics pipeline are within the graphics libraries OpenGL and Direct3D as shown in the figures 1.1 and 1.2

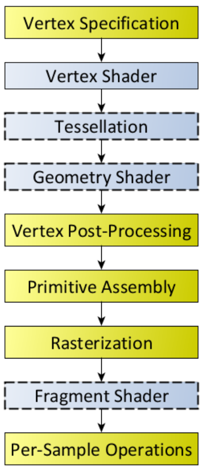


Figure 1.1: Graphics pipeline by OpenGL

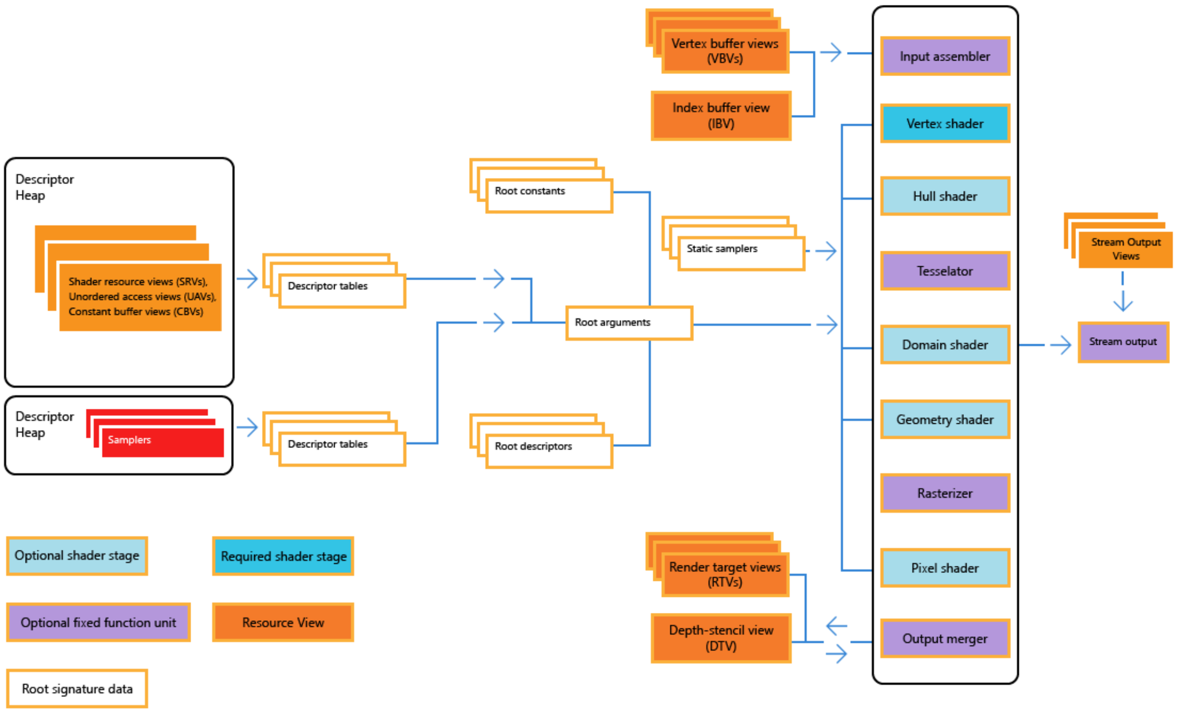


Figure 1.2: Graphics pipeline by Direct3D

Problem with debugging shaders in the graphics pipeline A shader is a program running on the GPU thereby mostly running as part of the graphics pipeline. [Khronos 2019b] The exception for this behavior is the compute shader which is independent from the graphics pipeline. [Khronos 2019c]

There are ways to debug shaders on the graphics pipeline, as shown in Section 2.1, but there are no general solutions for aiding the user in this debugging process. [Ciardi 2015]

While most CPUs are very broad in their functionality and support debugging by itself a GPU is more specialized in the way it functions and usually does not have the option to pause the code to enable inspections at runtime and does not even have access to a console or logger to write states into. [Fox 2017] As explained in more detail in Section 2.1 it is possible to enable debugging on the GPU with specialized drivers for specific hardware. [Nvidia 2019] [Microsoft 2016]

Existing approaches for compute shaders There are approaches which enable debugging of compute shader code by translating it from another language. The code is written in a language for the CPU which supports debugging with tools. It is run and debugged on the CPU while it is translated to the shader language to run as a compute shader on the GPU. See more in section 2.2.

Objective of creating a general solution for debuging shaders in the graphics pipeline The objective of this work is to create a general solution to enable debugging of shaders within the graphics pipeline. The goals this solution should fulfill are the following:

- The different methods shown to assist the programmer in debugging mentioned in Section 1 are usable.
- The solution is not dependent on using specific graphics cards or drivers.
- It is possible to switch between a mode where debugging is enabled and a mode where the shaders run as usual, so the program can run with the full performance and without interference of the debugger.
- The resulting output per render iteration of the debugger is close to the output of the program with the undebugged shader. It is close enough that the programmer can see what the rendering result without the debugger would look like. Errors like those resulting from the use of float variables with their inaccuracies are tolerable because there are tolerances within human perception where minimal changes in position or color within a rendered result do not matter. [Franz 2006]

- Performance is not a major requirement while running the debugger. It is possible to see the output and the values of the shader within each frame and iterate through the frames. It is not necessary to view the result in the speed of the final application while debugging. The programmer uses time to inspect the values within the shader which would not be possible with changing variables at high speeds like the 60 frames per second a usual graphics application has. [Christensson 2015] The debug application should fulfil acceptable response times for user interfaces. "10 seconds is about the limit for keeping the user's attention focused on the dialogue." according to [Nielsen 1993] .

2 Related Work

2.1 Existing methods for debugging shaders in the graphics pipeline

For debugging a shader program within the graphics pipeline there is the option to use workarounds to get the values of the variables within the code or by using special drivers provided by the producers of the hardware to get the option to debug on this hardware.

Manual debugging The manual way of debugging a shader is by creating outputs of the values within the shader program to see anomalies in their values. This can't be done by writing these values on the console or in a log file like it would be done in a CPU based application because, as mentioned in section 1, there is no access to the console or a logger within a shader program. The workaround used here is to return the values projected on the rgba-color values on the resulting image of the program. In this way the programmer can see the rough area in which the values are located on the direct output. The image can also be saved and inspected closer to get the exact values within the pixels of the resulting image. [Ciardi 2015]

Debugging with special drivers on certain hardware It is possible to install special drivers for certain graphics cards provided by their producers to enable debugging of shaders running on this hardware within dedicated environments. The two big suppliers of graphics hardware Nvida and AMD provide these debugging environments in the form of *Nvidia Nsight* and *GPU PerfStudio*. These tools can be included into different IDEs or downloaded as standalone applications. There are debugging tools for GPU debugging provided by other sources than the producers of the hardware themselves but for gaining access to the values in the hardware pipeline the drivers of this hardware have to implement specific debugging interfaces. [Microsoft 2016] Applying these tools enables the use of breakpoints and the inspection of variables within the shader code at runtime [*GLSL-Debugger*] or dumping the buffer into a file. [Microsoft 2019] The disadvantage of this method is that not all graphics cards are supported with such drivers and tools by their producer.

2.2 Approaches for debugging compute shaders on the CPU

As [Jukka 2012] states "Relatively little has been published about debugging of GPU programs in practice. Most of the best practice guides and lessons learned papers discuss GPU programming rather than debugging. Although exceptions exist". Among these exceptions are approaches to write compute shaders in other programming languages so that the version of the shader in the other language can be run on the CPU where it can be debugged. This version of the shader will then be translated to the actual shader language so it can run on the GPU with the full performance advantage. This enables the use of all tools the chosen language supports to aid debugging without depending on specific hardware or drivers.

The advantage compute shaders have in comparison with other kinds of shader programs is that they run independently of the graphics pipeline. They are executed by passing inputs and receiving the outputs like any method or program in regular CPU based languages. In the graphics pipeline some of the inputs and outputs would be handled by automated steps of the pipeline. The main difference between CPU based langages and GPU based langauges is that for the shaders passing inputs and receiving the output is done by writing into and reading from buffers.

There are projects like [*ILGPU*] where a special syntax is required for shaders within the CPU based language to enable the translation of this code as a compute shader to run on the GPU. Switch between running the code on the CPU where it can be debugged and running it on the GPU with optimal performance can be handled within the program.

Other projects like [*Campy*] provide a compiler translating the full codebase to run as a compute shader on the GPU. Here it is possible to write the code in the familiar language without having to adapt special syntax rules. To debug the code, it is run within a regular compiler for the language while it will run on the GPU when it is executed with the special compiler.

2.3 Translating shaders from other languages

To run code on the GPU usually the programmer has to use one of multiple low level languages specialized in this task. To enable programmers to run code on the GPU while being able to stay at the language they are accustomed to there are multiple solutions to translate different languages into these low level languages.

There are three different types of programming languages. The ones that are directly compiled ahead of time (AOT), the ones that are compiled to an intermediate language which then runs in a JIT(Just in Time) compiler and the ones that are running in an interpreter. For these three types there are different approaches to translate them to a shader language. [Turton 2017]

For all of these kinds of languages, it is possible to write a compiler translating them directly to the shader language. This is what [*GLSLplusplus*] does with the AOT compiled language C++ translating it to the shader language GLSL. In this project created to change the way of writing shader code the possibility of using this to debug the shaders written in C++ is mentioned. [*SharpShader*] is another example which uses the Language C# which is usually compiled to the intermediate language MSIL. In this example C# code is directly compiled to a Shader languagesHLSL or GLSL. In both cases the programmer can use his accustomed language with the limitation of having to write the code for the GPU within special syntax rules to be able to translate it to be translated into a shader for the GPU. An advantage that results of the direct translation is that variable names given in the written code can be retained in the shader code.

Another option with the use of a language that compiles to an intermediate language is to take the code in the intermediate language and further translate it into a shader language. Examples implementing this are [*ILGPU*] and [*Campy*] where the shaders are written in C#. This code is then compiled to the intermediate language MSIL. The MSIL code is further translated into the compute language CUDA which then runs on the GPU. Another example is [*SpirvNet*] which translates MSIL to Spirv. Spirv is a intermediate language for graphical shader code that can be interpreted as a shader on the GPU or translated further into other shader languages with tools such as [*SPIRV-cross*] which translates to GLSL. The translation in all of these cases is based on a uniform intermediate language which could be the result of multiple higher level languages. So by writing the translator for this intermediate language the option exists to write the shaders in different languages compilable to this intermediate language. The variable names are lost by compiling to the intermediate language which means that the variable names are not retainable in the shader code.

To get a code translated to a shader code it is also possible to chain multiple existing tools together to translate the language, in which the shaders should be written in, to the desired shader language. An example for this process can be seen in the way the Interpreted language R is translated through multiple steps to run on the GPU in the example "Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation" [Juan Fumero 2017]

3 Contribution

The proposed solution of this work is inspired by the approaches for compute shaders described in section 2.2. The code for the shaders is written in a source language which supports the different methods for debugging listed in 1.

To run the shader as usual on the GPU the shaders written in the source language is translated to the shader language and loaded on the hardware.

To enable the functionality while debugging, the linking of the shaders and the steps in between the shaders on the graphics pipeline, usually already provided by the graphics hardware, will be simulated in the other language.

3.1 Steps for translating the shader code

In most cases the access the attributes within a shader is done by getting its location from its name. [Lighthouse3d 2013] For being able to directly access the variables, a way to translate the shader from the source language without losing the variable names is chosen. As explained in section 2.3 by directly translating the language without compiling it to inbetween stages all kinds of languages languages can be chosen as source languages and the variable names can easily be retained.

Shaders within the graphics pipeline have a special way of being structured so the input and output within their respective stage is flowing correctly through the shader. For that reason it is not possible to translate any source code to the shader like described for the [*Campy*] example in section 2.2. For this reason the code written in the source language ment to be translated to a shader should mimic the structure the shader language provides. This means when the shader language provides certain structures for input or output or a specific point of entry into the code an equivalent for these has to be given in the written code.

The translation is done by converting the source language into a syntax tree from which the syntax will be extracted in the shader language. Occurances of variable types also have to be translated to their correct equivalent.

To be able to translate to a shader that is able to run sucessfully special patterns within shader code have to be minded. If the shader needs a header like GLSL shaders, which need the version declaration on top of the shader code, this has to be added manually in the translation process. Special syntax like the accessors in GLSL defining if a variable is a uniform or an attribute and which behavior the variables have in the shader which may not exist in the source language have to be compensated for. In chapter 4 as an example it is explained how to do this for translating C# to GLSL by the use of C# Attributes.

There are functionalities in the shader languages which can be used but should be avoided for performance reasons. For example the use of conditionals should be avoided and solutions where you multiply the result of the code within the condition with 0 or 1 depending on a mathematical representation of the condition should be prefered. [Holden 2013] To prevent the usage of such things the code could be replaced by the clean solution while translating or these cases are deliberately not translated in the first place.

3.2 Steps for simulating the graphics pipeline

To simulate the graphics pipeline or at least the parts of the graphics pipeline necessary to be able to run and thereby debug the shaders written in the source language on the CPU the shaders have to be written in a structure that they can be inserted and exchanged within the simulated pipeline. The simulated pipeline has to implement the steps usually running automated on the GPU inbetween the different shader stages.

The first step is to determine which kinds of shaders should be able to be simulated and therefore which parts of the pipeline as described in paragraph1 have to be implemented.

Instance of a shader within the source language A shader in the source language should be contained within a fixed structure like a class or at least within a single file. As already mentioned in section3.1 it is optimal to have the code of the shader mimic the structure of the shader in the original language. For each access point the regular shader has in the pipeline there is an equivalent in the simulation. The possibility to set variables as attributes or uniforms, the method with which the execution of the shader functionality is triggered and the option to access the outputs of the shader after it has been executed are given.

For the shader to be able to function on the CPU as on the GPU the types and functionalities existing in the shader language are implemented in the source language. For example there are the multiple forms of vector and matrix types based on the same base types. Also the methods and operators desired to be used of the shader language are implemented. The differende between value based and reference based behavior in the languages has to be considered in this step. [Magyar 2017]

Implementation of the graphics pipeline The way the shader is accessed and managed within the pipeline as well as the calculations in the steps between the shaders, as described in paragraph 1 are implemented. The different optional steps for optimizing in between the shader stages which can be enabled or disabled within the regular graphics pipeline like clipping, culling or the optional filters in the end are implemented if their use is desired. In the following the implementation of the basic steps is described.

The ordered list of the vertices as well as the attributes affiliated to them are saved in the pipeline.

To be able to set and get the attribute and uniform values by its variable names as it's done for actual shaders [Lighthouse3d 2013] a way to access the variables within the shader from a string containing the variable name is implemented.

To calcluate the vertex step, the vertex shader is executed for each vertex with the attribute values for the vertex being passed to the shader instance. The resulting vertex data is then gained as outputs of the shader instances.

The same process is done for each of the optional shader stages after the vertex step.

The perspective divida and the viewport transformation are calculated and applied to all vertices.

To implement the primitive assembly the vertices are put together to groups according to the primitive type which is defined within the pipeline.

A raster in the size of the resulting image is generated. within the raster for each position a fragmentlist is created. For each primitive and all positions in the raster overlaping it a fragment will be generated and saved in the corresponding fragmentlist within the raster. The attributevalues of the vertices the primitive consists of are interpolated according to the position of the resulting fragment and added to said fragment.

After all fragments are generated for each of them the depth and the stencil information is saved in new rasters and the fragment shader is executed with the attribute values being passed to the shader instance. The resulting fragment data is gained as outputs of the shader instances.

The color data of the fragments is finally added to a raster that can then be displayed as the renderresult.

4 Implementation

5 Conclusion

Fazit ziehen über das Projekt und die Arbeit. Welche Erkenntnisse wurden gewonnen? Was hat gut/schlecht funktioniert? Wurden die eigenen Erwartungen erfüllt oder nicht? War das Projekt erfolgreich?

Acronyms

CPU	Central Processing Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
JIT	Just in Time
AOT	Ahead of Time
MSIL	Microsoft Intermediate Language (also known as IL)

List of Figures

List of Tables

Bibliography

Campy. URL: http://campynet.com/ (visited on 08/24/2019).

Christensson, P. (2015). *FPS Definition*. URL: https://techterms.com/definition/fps# (visited on 08/22/2019).

Ciardi, Francesco Cifariello (2015). URL: https://computergraphics.stackexchange.com/a/101 (visited on 08/22/2019).

Collins, Harper (2014). *Collins English Dictionary – Complete and Unabridged, 12th Edition*. URL: https://www.collinsdictionary.com/dictionary/english/debugging (visited on 08/22/2019).

Fox, Alexander (2017). *MTE Explains: The Difference Between a CPU and a GPU*. URL: https://www.maketecheasier.com/difference-between-cpu-and-gpu/ (visited on 08/22/2019).

Franz, Dr. Maren (2006). *Die Subjektivität der Wahrnehmung*. URL: http://www.nlp-hh.de/img/Subjektivitaet_der_Wahrnehmung.pdf (visited on 08/22/2019).

GLSL-Debugger. *GLSL-Debugger*. URL: http://glsl-debugger.github.io/#features (visited on 08/23/2019).

GLSLplusplus. URL: https://github.com/AlexanderDzhoganov/GLSLplusplus (visited on 08/24/2019).

GPU PerfStudio. URL: https://gpuopen.com/archive/gpu-perfstudio/ (visited on 08/22/2019).

Holden, Daniel (2013). *Avoiding Shader Conditionals*. URL: http://theorangeduck.com/page/avoiding-shader-conditionals (visited on 08/27/2019).

ILGPU. URL: http://www.ilgpu.net/ (visited on 08/24/2019).

Jetbrains (2019). *Debugging Exceptions*. URL: https://www.jetbrains.com/help/rider/Debugging_Exceptions.html (visited on 08/21/2019).

Juan Fumero Michael Stuewer, Lukas Stadler (2017). "Just-In-Time GPU Compilation forInterpreted Languages with Partial Evaluation". In: URL: https://michel.steuwer.info/files/publications/2017/VEE-2017.pdf (visited on 08/24/2019).

Jukka, Julku (2012). "Debugging GPU code". In: URL: https://wiki.aalto.fi/download/attachments/70779066/jjulku_debugging_gpu_code_final.pdf?version=1&modificationDate=1357205536000&api=v2 (visited on 08/24/2019).

Kernighan, Plauger (1982). *The Elements of Programming Style, 2nd edition*. McGraw-Hill. ISBN: 978-0070342071.

Khronos (2019a). *Rendering Pipeline Overview*. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (visited on 08/27/2019).

– (2019b). *Shader*. URL: https://www.khronos.org/opengl/wiki/Shader (visited on 08/22/2019).

– (2019c). *Shader*. URL: https://www.khronos.org/opengl/wiki/Compute_Shader (visited on 08/22/2019).

Lighthouse3d (2013). *GLSL Tutorial – Attribute Variables*. URL: https://www.lighthouse3d.com/tutorials/glsl-tutorial/attribute-variables/ (visited on 08/27/2019).

Magyar, Szilard (2017). *All you need to know on by reference vs by value*. URL: https://www.freecodecamp.org/news/understanding-by-reference-vs-by-value-d49139beb1c4/ (visited on 08/27/2019).

Microsoft (2016). *Debugging GPU Code*. URL: https://docs.microsoft.com/de-de/visualstudio/debugger/debugging-gpu-code?view=vs-2019 (visited on 08/22/2019).

– (2018). *Pipelines and Shaders with Direct3D 12*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12 (visited on 08/27/2019).

Microsoft (2019). *Using automatic shader PDB resolution in PIX*. URL: `https://devblogs.microsoft.com/pix/using-automatic-shader-pdb-resolution-in-pix/` (visited on 08/23/2019).

Nielsen, Jakob (1993). *Usability Engineering*. Morgan Kaufmann. Chap. 5. ISBN: 978-0125184069.

Nvidia (2019). *Debugging Solutions*. URL: `https://developer.nvidia.com/debugging-solutions` (visited on 08/22/2019).

Nvidia Nsight. URL: `https://developer.nvidia.com/tools-overview` (visited on 08/22/2019).

SharpShader. URL: `https://github.com/Syncaidius/SharpShader` (visited on 08/24/2019).

SPIRV-cross. URL: `https://github.com/KhronosGroup/SPIRV-Cross` (visited on 08/24/2019).

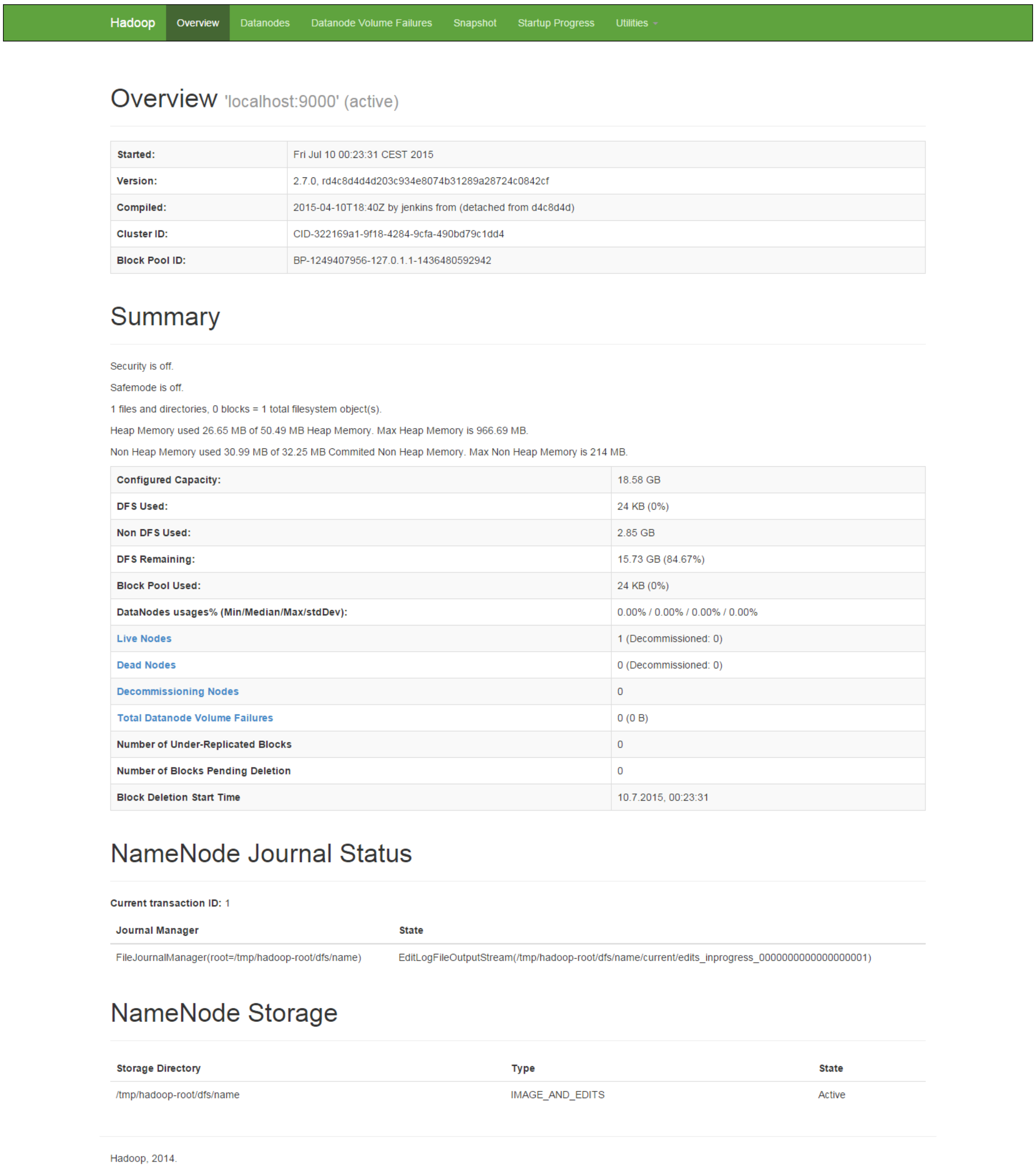
SpirvNet. URL: `https://github.com/Philip-Trettner/SpirvNet` (visited on 08/24/2019).

Turton, Lawrence (2017). *Interpreters, compilers, JIT & AOT compilation*. URL: `https://medium.com/@avelx/interpreters-compilers-jit-aot-compilation-753fb2f9014e` (visited on 08/24/2019).

Undo (2019). *What is Reverse Debugging, and why do we need it?* URL: `https://undo.io/resources/reverse-debugging-whitepaper/` (visited on 08/21/2019).

Appendix

A. Screenshot NameNode Web-Interface



Overview 'localhost:9000' (active)

Started:	Fri Jul 10 00:23:31 CEST 2015
Version:	2.7.0, rd4c8d4d4d203c934e8074b31289a28724c0842cf
Compiled:	2015-04-10T18:40Z by jenkins from (detached from d4c8d4d)
Cluster ID:	CID-322169a1-9f18-4284-9cfa-490bd79c1dd4
Block Pool ID:	BP-1249407956-127.0.1.1-1436480592942

Summary

Security is off.

Safemode is off.

1 files and directories, 0 blocks = 1 total filesystem object(s).

Heap Memory used 26.65 MB of 50.49 MB Heap Memory. Max Heap Memory is 966.69 MB.

Non Heap Memory used 30.99 MB of 32.25 MB Commited Non Heap Memory. Max Non Heap Memory is 214 MB.

Configured Capacity:	18.58 GB
DFS Used:	24 KB (0%)
Non DFS Used:	2.85 GB
DFS Remaining:	15.73 GB (84.67%)
Block Pool Used:	24 KB (0%)
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion	0
Block Deletion Start Time	10.7.2015, 00:23:31

NameNode Journal Status

Current transaction ID: 1

Journal Manager	**State**
FileJournalManager(root=/tmp/hadoop-root/dfs/name)	EditLogFileOutputStream(/tmp/hadoop-root/dfs/name/current/edits_inprogress_0000000000000000001)

NameNode Storage

Storage Directory	**Type**	**State**
/tmp/hadoop-root/dfs/name	IMAGE_AND_EDITS	Active

Hadoop, 2014.

C. DVD Inhalt

```
⊢ Anwendung/
|    – pom-xml                           ⇒ Maven POM Datei
|    ⊢ conf/                             ⇒ *.properties Dateien für Konfiguration
|    ⊢ src/                              ⇒ Quellcode Dateien
|    ⊢ target/
|    |    – Logfileanalyzer-1.0-SNAPSHOT.jar  ⇒ Ausführtbare JAR-Datei
|    |    ⊢ site/apidocs/                ⇒ JavaDoc für Browser
|
⊢ Literatur/                             ⇒ PDF Literatur & E-Books
⊢ Praesentationen/
|    – Abschlusspraesentation.pptx       ⇒ Präsentation vom 21. August 2015
|    – Abschlusspraesentation.pdf
|    – Kickoffpraesentation.pptx         ⇒ Präsentation vom 03. Juni 2015
|    – Kickoffpraesentation.pdf
|
⊢ Sonstiges/
|    – LineareRegression.xlsx            ⇒ Berechnung der linearen Regression
|
⊢ Latex-Files/                           ⇒ Editierbare LaTeX Dateien der Arbeit
     – bibliographie.bib                 ⇒ Literaturverzeichnis
     – dokumentation.pdf                 ⇒ Bachelorarbeit als PDF
     – dokumentation.tex                 ⇒ Hauptdokument
     – einstellungen.tex                 ⇒ Einstellungen
     ⊢ ads/                              ⇒ Header, Glosar, Abkürzungen, etc.
     ⊢ content/                          ⇒ Kapitel
     ⊢ images/                           ⇒ Bilder
     ⊢ lang/                             ⇒ Sprachdateien für LaTeX Template
```