



THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Shader Optimization and Specialization

Lewis Crawford



Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2022

Abstract

In the field of real-time graphics for computer games, performance has a significant effect on the player's enjoyment and immersion. Graphics processing units (GPUs) are hardware accelerators that run small parallelized shader programs to speed up computationally expensive rendering calculations. This thesis examines optimizing shader programs and explores ways in which data patterns on both the CPU and GPU can be analyzed to automatically speed up rendering in games.

Initially, the effect of traditional compiler optimizations on shader source-code was explored. Techniques such as loop unrolling or arithmetic reassociation provided speed-ups on several devices, but different GPU hardware responded differently to each set of optimizations. Analyzing execution traces from numerous popular PC games revealed that much of the data passed from CPU-based API calls to GPU-based shaders is either unused, or remains constant. A system was developed to capture this constant data and fold it into the shaders' source-code. Re-running the game's rendering code using these specialized shader variants resulted in performance improvements in several commercial games without impacting their visual quality.

Lay Summary

Video games are like flip-books. Every few milliseconds, a new picture appears on the screen, which gives the illusion of fluid motion. The faster these images can be drawn to the screen, the smoother the game looks and feels to play. With the graphical fidelity of games increasing, the complexity of the calculations required to draw these virtual worlds to the screen is also increasing.

The aim of this thesis's research is to find ways to automatically improve the performance of graphics calculations, to ensure that even games with complex graphics can continue to run smoothly. Modern video games all make use of specialized hardware known as a graphics processing unit (GPU), which is able to run small graphics programs known as shaders very quickly. This research explores several different techniques for optimizing these shader programs, so that they can run efficiently on different GPUs made by different companies.

Several ideas for optimizing these shaders are explored here. Firstly, the effect of different transformations performed on shader code is measured on different GPUs. Then, snippets of gameplay from various real-world games are examined to see whether there are any patterns in the data being fed to the shaders which can be exploited to make them run faster. By exploring these ideas, the aim is to motivate the building of tools that game developers can use to automatically improve the performance of their increasingly complex games with little manual intervention.

Acknowledgements

I would like to thank my supervisor, Professor Michael O'Boyle, for his patience and invaluable guidance throughout this project. I would also like to thank my parents for their support, and for proof-reading the final draft of this thesis. Finally, I would like to thank Erin for helping me through it all.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

- A Cross-platform Evaluation of Graphics Shader Compiler Optimization (Lewis Crawford, Mike O’Boyle - IEEE International Symposium on Performance Analysis of Systems and Software 2018)
- Specialization Opportunities in Graphical Workloads (Lewis Crawford, Mike O’Boyle - International Conference on Parallel Architecture and Compilation Techniques (PACT) 2019)

(Lewis Crawford)

Table of Contents

1	Introduction	1
1.1	Performance in Games	1
1.2	The Problem Domain	2
1.3	Research Contributions	3
1.4	Thesis Structure	4
2	Background	7
2.1	Introduction	7
2.2	3D Graphics Basics	8
2.2.1	Triangles	8
2.2.2	Textures	9
2.2.3	Shaders	12
2.3	Graphics APIs	14
2.3.1	The Purpose of Graphics APIs	14
2.3.2	Modern Graphics APIs	15
2.3.3	Trends Towards Lower-Level APIs	16
2.3.4	Shader Languages	17
2.3.5	Choosing OpenGL	18
2.4	Game Engines	18
2.4.1	The Role of an Engine	19
2.4.2	Game Engine Development	19
2.5	Programmable Pipeline Overview	21
2.5.1	Evolution of the Programmable Shader Pipeline	22
2.5.2	Modern Pipeline Layout	23
2.6	Transferring Data to the GPU	30
2.6.1	OpenGL Buffers	30
2.6.2	Uniforms	31

2.6.3	Uniform Buffers	33
2.6.4	Shader Storage Buffers	33
2.6.5	Textures	34
2.7	Summary	35
3	Related Work	37
3.1	Introduction	37
3.2	Evolution of Programmable Shaders	37
3.2.1	Early Shader Conceptualization	37
3.2.2	Early Rendering Hardware	38
3.2.3	Multi-Pass Shading	39
3.2.4	Programmable Shaders on GPUs	39
3.2.5	GPGPU Programming Emerges	40
3.2.6	New Shader Types	40
3.3	Shader Simplification	41
3.3.1	Level of Detail	42
3.3.2	Shader Level of Detail	42
3.3.3	Surface Signal Approximation	43
3.3.4	Altering Computation Rates	43
3.4	Value-based Optimizations	44
3.4.1	Constant Propagation and Folding	45
3.4.2	Branch Prediction	45
3.4.3	Value Prediction	46
3.4.4	Run-time Specialization and JIT Compilation	47
3.4.5	Value Profiling	50
3.4.6	Value-Based Optimizations for GPUs	51
3.5	Energy Efficiency in Mobile Games	52
3.5.1	CPU DVFS for Software Rendering	52
3.5.2	The Allure of Quake II	53
3.5.3	Closed-Source Workloads on GPUs	54
3.5.4	DVFs for CPU, GPU, and Memory	55
3.5.5	Dynamically Varying Frame Rates	56
3.5.6	Avoiding Overdraw	57
3.5.7	Variable Floating-point Precision	57
3.6	GPU Debugging and Profiling	58

3.6.1	Current Industry Tools	58
3.6.2	GPU Debugger Research	59
3.6.3	GPU Profiling & Performance Estimation Research	60
3.7	Summary	60
4	Compiler Optimizations for Individual Shaders	63
4.1	Introduction	63
4.2	Motivating Example	64
4.3	Example Optimizations	66
4.3.1	LunarGlass Optimization Framework	66
4.3.2	Additional Unsafe Optimizations	67
4.3.3	Artefacts	69
4.4	Benchmark Characteristics	71
4.4.1	Benchmarks within GFXBench 4.0	71
4.4.2	Extracting Shaders	74
4.4.3	Deduplicating Shaders	75
4.4.4	Shader Characteristics	76
4.5	Timing Tools and Experimental Setup	80
4.5.1	Shader Execution Enviroment	80
4.5.2	Vertex Shader Generation	82
4.5.3	Hardware	85
4.6	Timing Results	86
4.6.1	Overall Performance	86
4.6.2	Best Static Flags	86
4.6.3	Per-shader Results	87
4.6.4	Per-Flag Results	89
4.6.5	Summary	93
4.7	Conclusion	93
5	Analysis of Potential Optimizations Within Shader Pipelines	95
5.1	Introduction	95
5.2	Motivating Example	96
5.3	Example Optimizations	97
5.3.1	Dead	99
5.3.2	Movable	100
5.3.3	Constant	101

5.3.4	Constant Foldable	101
5.4	Techniques for Detecting Potential Optimizations	102
5.4.1	Dead Code/Data Analysis	102
5.4.2	Movable & Constant Code Detection	103
5.4.3	Dynamic Trace Analysis	104
5.5	Benchmark Games	105
5.6	Static Analysis Results	107
5.6.1	Static Dead Code and Data	107
5.6.2	Statically Movable Code	110
5.7	Oracle Study on Constant Input Data	112
5.7.1	Constant Uniforms	112
5.7.2	Constant Inputs	114
5.7.3	Constant Textures	116
5.7.4	Oracle Study Summary	117
5.8	Trace Analysis Results	118
5.8.1	Constant Uniform Values	118
5.8.2	Redundant Uniform Updates	118
5.9	Timing Tests	120
5.10	Conclusion	122
6	Optimizations Within Full Execution Traces	125
6.1	Introduction	125
6.2	Benchmark Games	126
6.2.1	Selecting Games	126
6.2.2	Capturing Traces	130
6.3	Tools Developed	133
6.3.1	Overview	133
6.3.2	Tracking Shaders and Programs	135
6.3.3	Tracking Uniform Data	140
6.3.4	Tracking UBOs	143
6.3.5	Creating Specialized Shaders	144
6.4	Timing Techniques	147
6.4.1	Full-trace timings	148
6.4.2	Repeated Frame Timings	152
6.5	Performance Results on Whole Execution Traces	153

6.5.1	Constant Data	154
6.5.2	Timing Results	155
6.6	Conclusion	163
7	Conclusion	165
7.1	Summary	165
7.2	Critical Evaluation	167
7.3	Directions for Future Work	169
	Bibliography	171

Chapter 1

Introduction

1.1 Performance in Games

Real-time graphics is a field with strict performance requirements. Graphical applications such as games work similarly to flip-books – a series of still images (frames) are displayed quickly to give the illusion of motion. The faster the frames are drawn (rendered), the smoother the motion appears.

For PC games, rendering with at least 60 frames-per-second (FPS) is considered standard, especially when quick reaction-times are required[1][2], so games must update and render their entire simulated world within 16ms. Emerging applications such as virtual reality require 90FPS, and high-end gaming monitors allow games to render at 120 or even 144 FPS if the PC hardware can keep up.

With such tight timing requirements, significant developer effort is necessary to optimize performance across a wide variety of PC, mobile, or console hardware. Tools to aid or automate portions of this optimization process would be valuable, as even shaving off fractions of a millisecond may allow the game to hit the target frame rate without requiring additional significant developer efforts.

Despite graphics being a burgeoning industry projected to be worth \$215 billion by 2024[3], performance optimizations within computer graphics systems receive relatively little academic attention. Many papers exist exploring new graphics algorithms, GPU architecture improvements, or optimizations for general-purpose compute applications, but work exploring systems-level optimizations tends to focus purely on energy-efficiency, rather than run time. This thesis examines a variety of real-world games and graphics benchmarks, to quantify and explore the opportunities for creating systems to aid in automatically improving the run-time performance of games.

1.2 The Problem Domain

The Graphics Processing Unit (GPU) is a specialized piece of highly parallel hardware designed to accelerate rendering in games. It is programmed using pipelines of small single-program multiple-data (SPMD) graphics kernels called *shaders*. Thousands of instances of these SPMD shaders can run in parallel on different input data (e.g. one per pixel) before passing their output to the next stage of shaders in the pipeline, and eventually outputting pixel colours to the screen. To set up these shader pipelines and control the GPU, games use a hardware-agnostic graphics API such as OpenGL[4]. Different hardware-specific implementations of these graphics APIs are supplied by GPU vendors within their drivers.

GPU drivers are filled with performance-boosting heuristics. GPU vendors often patch drivers shortly after major big-budget game releases with game-specific optimizations. One of the goals for newer lower-level graphics APIs like Vulkan[5] and DirectX 12[6] is to provide developers with more fine-grained and explicit control of the GPU. This aims to enable developers to create their own highly tuned application-specific optimizations, rather than requiring GPU vendors to build fast-paths through their drivers for each major game release.

Low-level APIs offer lower overheads, increased parallelism, more predictable cross-platform behavior, and finer-grained control at the expense of removing many driver heuristics. However, developers have access to application-specific knowledge that provides optimization opportunities beyond those possible within the GPU driver.

The GPU driver must compile shader code within tight time-constraints, as shaders may be dynamically compiled in the middle of a running game. Pausing too long in a compilation stage may cause annoying stuttering and hitching in the game's previously smooth frame rate. Even if games attempt to avoid these sudden stalls during the gameplay by batching shader compilation during a single loading stage, the GPU driver may need to compile and link hundreds of shader pipelines quickly to avoid frustratingly long loading times for users.

If shader optimizations were performed by developers ahead of time during an offline build-process, they may be able to perform operations that requires significant time and resources to apply. Iterative compilation is one such optimization technique explored within this thesis, which is well outside the scope of the strict real-time requirements of shader compilers in graphics drivers. Analysis of these offline shader optimizations is contained within [Chapter 4](#).

As well as having additional time to perform long-running optimization techniques, additional game-specific information can be exploited at this early offline stage too. Such knowledge may be gleaned by static analysis of shader source code, or by examining execution traces of games to exploit data-flow patterns within them. It has been shown that over 99% rendering data is reused between frames[7], but is impractically large to keep in cache. This thesis explores how game-specific knowledge can help drive automated specialization and optimization techniques.

1.3 Research Contributions

This research explores several optimization techniques suitable for improving games' real-time rendering performance as part of an automated offline process. Performance analysis and workload characterization are also performed for twenty five real-world commercial games, as well as the well-known GFXBench graphics benchmark[8].

In [Chapter 4](#), the efficacy of several traditional compiler optimizations, such as loop unrolling and arithmetic reassociation, is explored by applying them to the source-code of shaders. Using iterative compilation to determine the optimal sets of compiler optimization passes on different GPU hardware, speed-ups of 4-13% can be achieved on some complex fragment shaders[9] from GFXBench. This work builds on an existing open-source shader optimization tool[10], and extends it to include further optimization passes such as unsafe floating-point arithmetic re-ordering.

Subsequent work in [Chapter 5](#) extends the scope of these fragment shader optimizations to explore whole shader pipelines, and code-motion opportunities between the GPU and CPU, or between different shader pipeline stages[11]. Extensive static analysis is performed on shader source-code from eight commercial games, as well as analysis of data patterns in execution traces recorded from them.

Techniques for extracting shaders from these games are explored, and techniques are presented for quantifying what proportion of shader code and data are unused, movable, or constant-foldable. An oracle study also explores how the proportion of specializable code changes based on knowing data from different input sources to be constant at run-time.

Execution traces are analysed by extending an open-source trace tool[12]. This analysis explores what proportions of values passed from the CPU to GPU remain constant throughout a game's lifetime, and how many updates to this data are redundant. The results of these techniques determine that many games have significant amounts

of data that is either unused or remains constant at runtime, which can be used for aggressive shader specialization (as shown in [Chapter 6](#)). Exploiting this data can lead to an automatic $\sim 5\%$ improvement in rendering entire frames of numerous games with no visual degradation or accuracy trade-offs (judged by manually inspecting the output images before and after the optimization, and finding no visually perceptible differences).

As well as the above analysis and optimization tools described above, this research also involved the creation of several timing and microbenchmarking tools for measuring the performance improvements of the above techniques. These tools explored timing individual fragment shaders, individual shader pipelines, and fully rendered frames extracted from execution traces.

1.4 Thesis Structure

Below is a brief guide to the contents of the subsequent chapters.

Chapter 2: Background Real-time graphics is a vast field, whose explanation is beyond the scope of this document, and is better suited to a textbook[13]. However, a basic understanding of typical graphics pipelines, and the role of shader within them is necessary to understand the optimizations performed in subsequent chapters. This background section aims to provide the reader with sufficient details about shader-based rendering pipelines on GPUs, graphics APIs, and the ecosystems around them. No rendering techniques or algorithms are explored here, but the general concepts of 3D triangle-based models, 2D textures, and shader program pipelines will all be explained.

Chapter 3: Related Work Although few systems-level papers directly in the field of optimizing games' run-time performance exist, work on optimizing energy efficiency in games features many similar techniques. This chapter explains the evolution of the programmable shader pipeline, and summarizes work from adjacent fields of optimizing game power consumption, and shader simplification techniques.

Chapter 4: Compiler Optimizations for Individual Shaders In this chapter, individual fragment shaders are extracted from the GFXBench 4.0 benchmark suite[8], and

undergo a series of source-to-source compiler optimizations. Using iterative compilation, the best set of optimizations are selected for different shaders on different GPU hardware, and the merits of different compiler optimizations are discussed.

Chapter 5: Analysis of Potential Optimizations Within Shader Pipelines Extending the previous chapter's work beyond individual fragment shaders, this chapter examines shader pipelines extracted from execution traces of real games. Numerous static analysis passes are introduced to quantify shader specialization opportunities. An extensive oracle study also explores how these specialization opportunities increase if run-time analysis can determine a shader's inputs from various sources to be constant. Dataflow patterns within game execution traces are also examined for eight games.

Chapter 6: Optimizations Within Full Execution Traces Motivated by the previous chapter's data analysis, tools for analysing and modifying execution traces from 17 commercial games are developed. Constant data is folded into the source-code of all shaders in the trace, and the performance for rendering entire frames is measured, resulting in speed ups for several games.

Chapter 7: Conclusion The findings of the prior three chapters are summarized, and potential future directions of research are discussed.

Chapter 2

Background

2.1 Introduction

The field of real-time graphics combines aspects of science, art, and technology. A rendering engineer must interpret findings from optics and photo-chemistry about light and its interactions with the environment, as well as its perceptual interpretation as visible colours. Using this knowledge, they must come up with clever abstractions and simplifications that allow 3D virtual worlds to be conveniently authored by artists, and efficiently simulated by computer hardware. To achieve the smooth frame-rate of 60 frames per second (FPS) that is viewed as standard within the PC gaming market, the entire world must be simulated and drawn to the screen (rendered) within 16ms.

New rendering techniques are constantly being developed that make real-time graphics applications such as computer-games more beautiful, more physically accurate, easier for artists to iterate on, or improve their performance on current hardware. At the same time, the hardware in graphics processing units (GPUs) is also evolving to allow game developers to run these graphics algorithms more quickly, more flexibly, or more power-efficiently. Standardized graphics APIs act as an intermediary between the ever-shifting hardware from different GPU vendors, and the developers creating games that need the hardware to accelerate their graphics algorithms.

Although the specific algorithms, hardware, and APIs are always changing, modern 3D graphics rendering can generally be thought of as 3D models passing through a pipeline of highly data-parallel calculations, which result in a 2D grid of colours being sent to the screen. The purpose of the GPU is to accelerate these data-parallel calculations, and efficiently pass data between different stages in this rendering pipeline. As time has gone on, more pipeline stages have been added, and the types of calcu-

lations possible at each pipeline stage have become more flexible. This flexibility has allowed for the emergence of general-purpose GPU computations (GPGPU), which allows other non-graphics calculations to benefit from the same data-parallel acceleration hardware. Many computer games also utilize this GPGPU capability for parts of their rendering calculations that do not fit into the traditional pipeline model.

This chapter outlines the basics of the 3D graphics pipeline. However, it will not cover any specific rendering algorithms or techniques, as these form a vast rapidly evolving field that could fill several textbooks[13]. Instead, the focus here is on the main pipeline stages used in graphics calculations irrespective of the specific algorithms being implemented. The way that data flows between pipeline stages, and between the CPU and GPU, will also be covered.

2.2 3D Graphics Basics

This section describes the basic shape of the 3D rendering problem, and outlines the parallel pipeline model that most real-time graphics applications use to solve it quickly on the GPU. The aim is to introduce how 3D worlds are typically represented, and how shader programs on the GPU are used to render them.

2.2.1 Triangles

Rendering is the process of taking the representation of a 3D virtual world, and transforming it into a 2D grid of coloured pixels. Although there are many ways a 3D world can be represented, real-time rendering applications almost invariably describe the world's geometry as a set of 3D meshes made up of triangles (see [Figure 2.1](#)). Triangle meshes can be used as simple approximations to describe arbitrarily complex shapes. A 2D billboard can be made up of 2 triangles, or a flat cuboid wall requires only 12 triangles, but a highly detailed model of a game's protagonist may contain tens-of-thousands of minuscule triangles to carefully capture the precise curvature of their facial shape or equipment when viewed close to the in-game camera.

Triangles provide a very simple primitive that can be used to approximately describe much more complex geometry, and they have numerous benefits. Triangles are guaranteed to be convex, which is a useful property during the rasterization process. Rasterization is where the surface of the triangle that will appear on screen is filled in with pixels. Values calculated at the 3 vertices of a triangle can also be smoothly

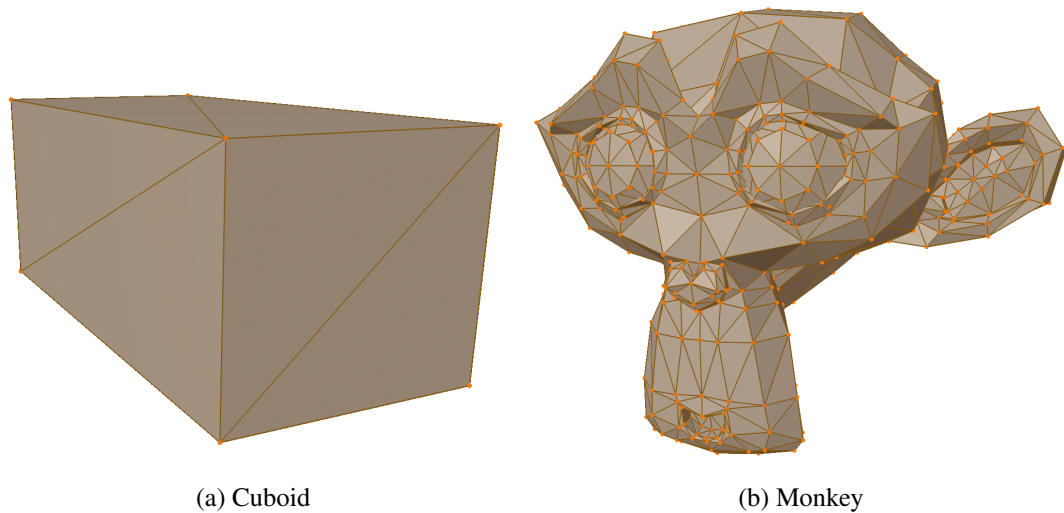


Figure 2.1: Examples of how many triangles can be used to form arbitrarily complex meshes of different shapes.

blended across every point of the triangle's surface using linear interpolation, meaning many calculations only need to occur once per vertex, rather than once per pixel within the triangle's surface.

2.2.2 Textures

As well as using triangle meshes to describe 3D models, another common resource used within games is textures. These textures are often 2D images which can be wrapped around the surface of a 3D mesh similar to applying wrapping paper to a complex-shaped present. They can be used to provide additional surface details to the model without having to add in numerous extra triangles to the mesh.

A texture's image data may be thought of as a 2D array of pixel colours stored as red-green-blue (RGB) floating-point colours. However, textures need not hold only a simple colour value, and can be used to represent surface normals¹, shadows, heights to displace waves, maps of areas affected by wind, snow, blood, or footprints, or physical properties about how the surface emits or reflects light. 2D textures are the most common use case, but it is also possible to use 1D textures for look-up tables or histograms, and 3D textures for volumetric effects like light-rays, fog, smoke, or fire.

The way a 2D texture gets wrapped around a 3D triangle mesh is described by

¹Normals are vectors orthogonal to a surface that describe which direction it faces. Ubiquitous within graphics, they help determine the angle light hits a surface for reflection or refraction.

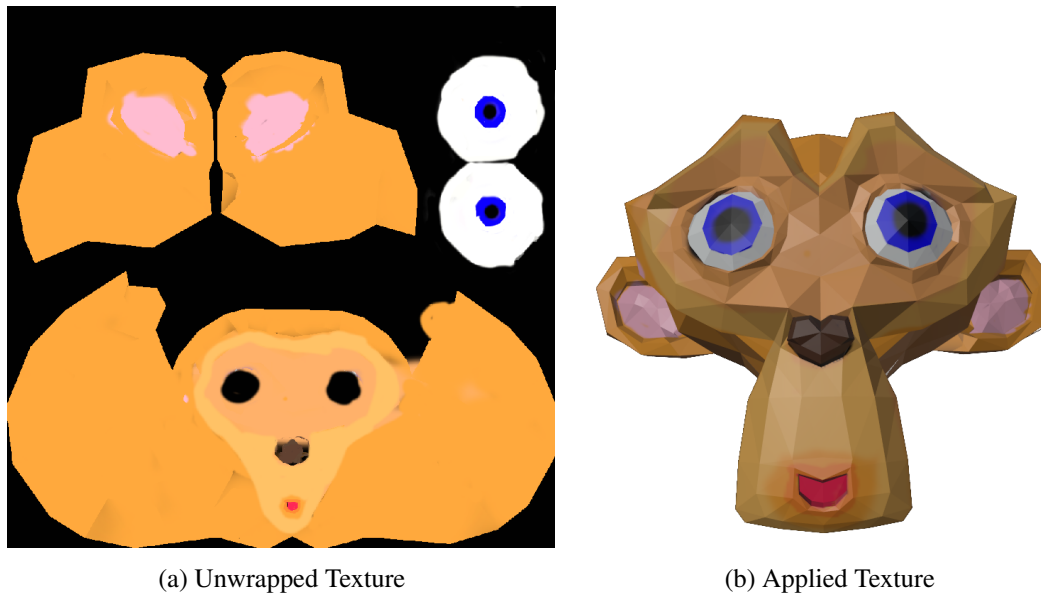


Figure 2.2: Unwrapped texture examples

UV-mapping. Each point on the 2D texture grid can be mapped to 2 coordinates, (U, V). These are floating point numbers between 0.0 and 1.0, representing the horizontal and vertical components of the texture grid. 3D modelling software packages have methods for unwrapping all the triangles of a 3D mesh and mapping them to 2D UV coordinates on a flattened texture map (see [Figure 2.2](#)). The algorithms used to perform this unwrapping try to balance several different concerns, such as ensuring larger triangles in the mesh are mapped to larger areas of the 2D texture to increase their pixel-density[14].

It is also often desirable to have adjacent triangles in the 3D model be adjacent to each other in the flattened mesh. This makes it easier for artists to visualize which parts of the model's texture they are editing, and allows a smoother blend between the colour values between adjacent triangles. If a model is unwrapped poorly, it is possible for noticeable artefacts to occur such as visible UV-seams where triangles adjacent on the 3D mesh had jarringly different colours in the unwrapped version, or when the background colour of the texture bleeds onto the model instead of the artist's intended colour. Numerous different UV unwrapping algorithms exist, many of which allow for manual interventions by artists to choose good splitting points and make the seams easier to hide. No matter which algorithm is used, however, the end result is a set of floating point UV-values between 0.0 and 1.0 for each vertex of each triangle.

Conceptually, textures are quite similar to 2D arrays, but they are accessed using floating-point UV coordinates between 0.0 and 1.0 as indices, as shown in [Figure 2.3](#).

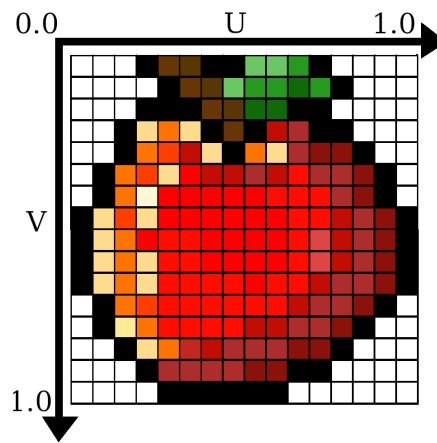
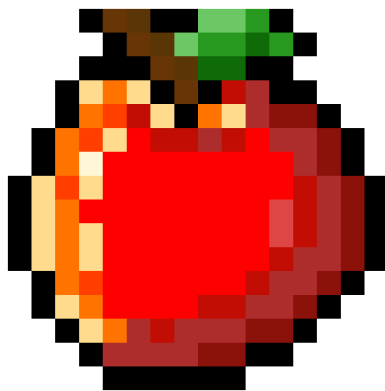


Figure 2.3: UV coordinates use floating point numbers between 0.0 and 1.0 to index into a texture image

These floating-point indices can address more precise positions than the coarse discrete integer grid indexing used in traditional 2D arrays. This precise sub-pixel indexing allows hardware-accelerated filtering algorithms to smoothly blend the resulting colour values of neighbouring pixels. Examples of different texture filtering methods are shown in [Figure 2.4](#). GPUs typically have portions of memory and cache-hierarchies specifically for performing texture look-ups, and dedicate significant proportions of their on-chip area to specialized hardware to fetch, decompress, and filter the texture values quickly.



(a) Nearest Neighbour Filtering



(b) Bilinear Filtering

Figure 2.4: Examples of sampling 16x16 pixel texture to across a 1024x1024 screen with different filter algorithms. Nearest neighbour filtering produces crisp pixel boundaries, and is good for preserving low resolution pixel art. Bilinear filtering produces smoother results, and is better for rendering more realistic higher resolution textures.

2.2.3 Shaders

In addition to the 3D triangle meshes used to describe a scene's geometry, and the 2D textures describing surface colours and properties, the final main element in graphics applications is the GPU shader programs. These shaders are often small pieces of code, no more than a few hundred lines long, and are used to calculate each triangle's position on screen, and each pixel's final colour.

Shaders may be written directly by technical artists to describe precise visual effects. They may also be written by rendering engineers as part of a multi-pass rendering pipeline, with tunable parameters exposed for artists to customize. Shaders can also be auto-generated, either by cross-compiling from shaders from different languages, or as a target for a higher-level language, or even a node-based material description system.

Shaders run in parallel at different stages of a pipeline, which consists of up to five different stages, the two most important of which are the vertex and fragment shaders. The pipeline's shape, and other optional shader stages are discussed in [Section 2.5](#).

Vertex Shaders

Vertex shaders run once for every vertex in a 3D triangle mesh. As with all shaders, they may perform arbitrary calculations, but vertex shaders' primary purpose is to determine where the input vertex from the 3D model should appear within the viewport of the camera. The position and orientation of both the object being rendered and the camera can be provided to the shader, and it can usually calculate the resulting viewing position using some simple matrix multiplication.

More complex per-vertex calculations may also occur, such as skeleton-based animation, or some form of height, bump, or displacement mapping to offset the model's vertices for effects like waves, mountainous terrain, lava bubbles, or noisy ripples. [Figure 2.5](#) shows an example vertex shader performing height-map-based displacement.

As well as the position data, the vertex shader can also output other information such as texture UV coordinates, normals, or colours associated with a particular vertex. Each vertex's results can be calculated independently, so vertex shaders run in parallel for every vertex of a mesh. As there may be thousands of triangles in a mesh, this makes vertex shading a highly data-parallel operation perfect for GPU acceleration.

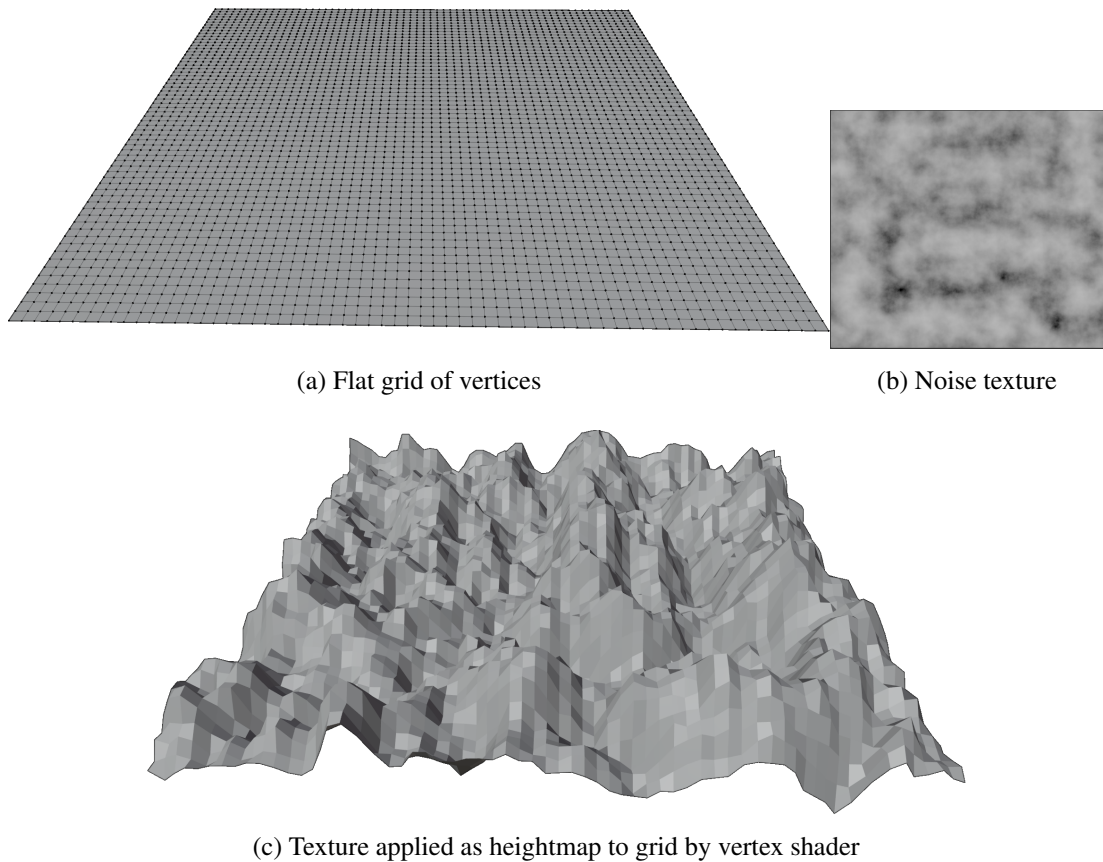


Figure 2.5: Example of vertex shader transforming a flat grid of vertices into randomized terrain using a Perlin noise texture as a heightmap

Fragment Shaders

Fragment shaders are the final stage in the programmable graphics pipeline. They run in parallel once for every visible pixel on the surface of each triangle, and output the final colour to display at that point on the surface. Values calculated in the vertex shader for each vertex get linearly interpolated across the triangle's surface, and fed in as inputs to the fragment shader. This means that any surface normals or texture UV coordinates outputted from the vertex shader gets blended together with the triangle's other corners, allowing a smooth transition of values across the triangle's surface even if it contains many pixels.

Fragment shaders may perform complex physically-based lighting calculations to achieve realistic effects, or may opt for more cartoon-style colouring techniques like cell-shading depending on the art-style of the game, and the processing power of the target hardware it is running on. An example fragment shader using a simple phong-

lighting[15] algorithm to shade a spherical mesh's surface is shown in [Figure 2.6](#).

As well as determining the pixel colours or properties for individual 3D models being rendered, fragment shaders may also be used for various full-screen effects such as motion-blur[16][17], colour-mapping[18], or screen-space ambient occlusion[19]. They provide a flexible and efficient way of calculating per-pixel colours in parallel using arbitrary lighting algorithms.

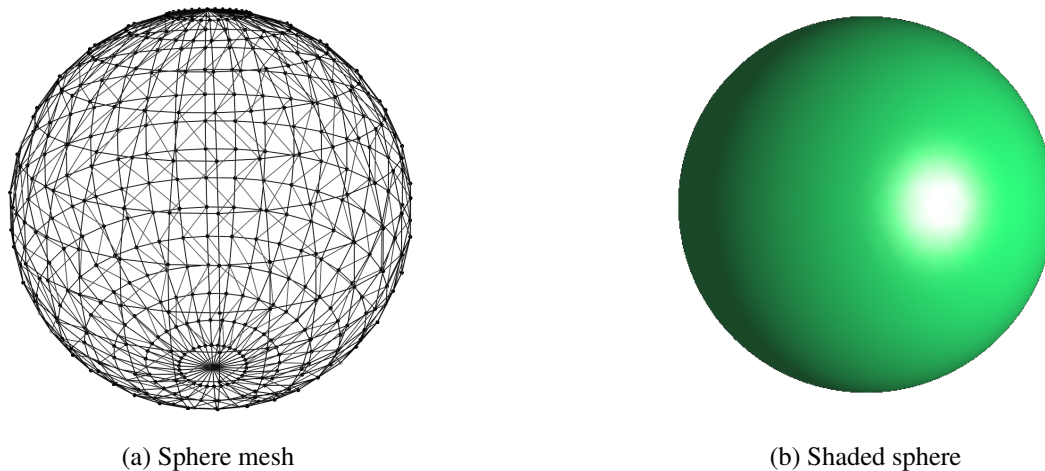


Figure 2.6: A spherical triangle mesh before and after a simple phong-lighting fragment shader has been applied to select its pixel colours

2.3 Graphics APIs

This section explores the role graphics APIs play within the graphics ecosystem. OpenGL terminology will be used throughout this thesis, as it is a cross-platform API utilized by all the Linux and Android-compatible software examined as benchmarks in subsequent chapters. However, the broad concepts are the same across all the modern graphics APIs described in [Section 2.3](#).

2.3.1 The Purpose of Graphics APIs

Applications such as computer games that make heavy use of real-time graphics are often large and complex pieces of code. Games targeting individual consoles such as the Nintendo Switch or PS5 may be able to heavily specialize their code to optimize performance on the target hardware. However, most console games target multiple

different consoles, so have to adapt to several different hardware configurations of varying capabilities. For PC or mobile games, thousands of hardware configurations exist with GPUs from numerous vendors (in 2015, there were over 24,000 distinct android devices[20]). Unified graphics APIs exist to allow real-time graphics code to be portable between different GPUs from different vendors.

The spectrum of GPU hardware that graphics APIs must account for is quite varied. Vendors such as NVIDIA and AMD ship discrete GPUs that are separate devices containing their own on-board memory, and are controlled via the PCIe bus. Many Intel CPUs contain embedded GPUs built directly into the CPU chip. Mobile devices using GPUs from ARM, Qualcomm, or Imagination also feature embedded GPUs that can make use of a shared memory-pool and caching system with the CPU. Discrete GPUs are generally much higher performance than the smaller embedded GPUs, but are more power-hungry as a result.

Even within devices of the same form-factor from the same vendor, GPUs may vary drastically in terms of the internal instruction-set they use, the number of physical shader/compute execution cores, and the size and types of caches and memory available. Graphics APIs must abstract over all these differences to make developing portable rendering software tractable, while still exposing enough hardware differences to allow some degree of platform-specific performance optimization.

2.3.2 Modern Graphics APIs

Graphics APIs are designed as an interface between rendering applications and the GPU hardware. They are implemented partly within the operating system, and partly within the GPU drivers. Vulkan [5], OpenGL[4], and its mobile counterpart OpenGL ES[21], are open-standard cross-platform graphics APIs. They are specified by the Khronos Group[22], which contains members from all GPU vendors, as well as various large game developers and software companies[23].

Microsoft's own DirectX graphics APIs[24] are only available on Windows machines. However, as Windows covers the vast majority of the PC-gaming market-share[25][26], DirectX is often used as the de-facto API for games that do not need to be cross-platform. Apple formerly used OpenGL and OpenGL ES for all its devices, but has recently deprecated them[27] in favour of its new Metal API[28]. A map of current graphics API compatibility on different operating systems is portrayed in [Table 2.1](#).

API	Windows	Linux	Mac OSX	iOS	Android
OpenGL	✓	✓	✓	-	-
OpenGL ES	-	-	-	✓	✓
DirectX 11	✓	-	-	-	-
DirectX 12	✓	-	-	-	-
Metal	-	-	✓	✓	-
Vulkan	✓	✓	-	-	✓

Table 2.1: Compatibility of different operating systems with different graphics APIs.

2.3.3 Trends Towards Lower-Level APIs

Historically, OpenGL and DirectX were designed as interfaces to GPUs with fixed-function hardware, and have evolved gradually to adapt to the more flexible programmable shader pipelines available today (see [Section 2.5](#)). These older APIs aimed to abstract away many complicated aspects of GPU programming from application developers. Such abstractions included assuming a single-threaded application, heavily validating inputs, and leaving the exact timings of command execution and GPU memory synchronization be decided by GPU drivers. This simplified many complexities for application developers, and allowed GPU vendors flexibility in implementing how and when commands and data would be submitted to their GPUs.

However, these abstractions had several downsides, such as making it difficult for developers to reason about and optimize their games' performance. Only having a single thread to interface with the GPU via API calls that often incurred unnecessary validation overheads meant that the CPU rendering thread often became a performance bottleneck, leaving the GPU underutilized.

Recently, the games industry has been increasingly favouring lower-level graphics APIs providing better parallelism and more explicit control of the target hardware[29][30]. Microsoft's DirectX 12 API[6], Apple's Metal API[28], and the Khronos Group's cross-platform Vulkan[5] API all aim at solving the performance and parallelism deficiencies of OpenGL and DirectX 11, and provide game-developers more fine-grained control of the GPU.

2.3.4 Shader Languages

Every graphics API consists of a set of CPU-side functions called by the host application, and a method of writing shaders to run on the GPU, typically using a C-like language. Microsoft DirectX shaders are written in HLSL (High Level Shading Language)[31], and are compiled down to DXBC byte-code[32] for DirectX 11. For DirectX 12, they are compiled into DXIL[33], an LLVM-like intermediate representation (IR)[34]. Metal uses the Metal Shading Language[28], which is based on C++14[35], and is also compiled into an LLVM-like IR. A summary of the shader languages compatible with each API is shown in Table 2.2.

API	GLSL	HLSL	SPIR-V	Metal SL
OpenGL	✓	-	(4.6 Onwards)	-
OpenGL ES	✓	-	-	-
DirectX 11	-	(Via DXBC)	-	-
DirectX 12	-	(Via DXIL)	-	-
Metal	-	-	-	(Via LLVM IR)
Vulkan	(Via SPIR-V)	(Via SPIR-V)	✓	-

Table 2.2: Compatibility of graphics APIs with different shading languages and intermediate representations (IRs)

OpenGL and Vulkan shaders are usually written in GLSL (OpenGL Shading Language) [36], but HLSL[31] can also be used in Vulkan. Vulkan shaders are compiled into SPIR-V[37], a high-level IR. OpenGL applications typically use the GLSL source-code directly as an input, and all shader compilation occurs within the OpenGL driver. In version 4.6[4], OpenGL also accepts SPIR-V shaders, but most games still provide the GLSL source strings directly. This feature of OpenGL allows the shader source code to be intercepted and extracted from the games used as benchmarks in subsequent chapters.

Because Microsoft Windows accounts for over 96% of the PC gaming market share[26], it is common for shaders to be written using HLSL for DirectX, and then cross-compiled into GLSL to run on OpenGL-based platforms. As a result, many of the shaders extracted from OpenGL-based games throughout this thesis had already undergone some degree of automated transformation.

2.3.5 Choosing OpenGL

OpenGL was selected as the API to examine within this thesis for several reasons. Firstly, it is compatible with all desktop platforms. On mobile platforms, shaders require only minor transformations to conform to OpenGL ES. This enables a wider variety of hardware and software options when benchmarking shader optimizations.

Another benefit of OpenGL is the fact it provides shader source-code directly to the API, rather than using a pre-compiled bytecode or IR. This allows the source-code to be intercepted and extracted from closed-source games for analysis throughout this thesis. The fact that GLSL shaders are often the subject of automatic cross-compilation from HLSL[38][39][40] also means that optimizations discovered here could be added to these pre-existing transformation pipelines.

Although some newly released PC titles use Vulkan for cross-compatibility with Linux and other platforms now, transitioning to a new graphics API requires major engineering commitment[41][42][43][44][45], so OpenGL is still in use. There is also a vast library of OpenGL-based games already on the market, which far outnumber the few recently released Vulkan-based titles. As such, OpenGL-based benchmark timings will be representative of the majority of games for years to come.

Vulkan's tools ecosystem is still rapidly evolving[46], whereas OpenGL already has many well-established tools built around it[47][48]. Execution tracing[12] and shader compilation[10] tools were used extensively throughout this research, and the pre-existing open-source tools for these tasks for OpenGL provided a useful starting point.

OpenGL was selected for its wide cross-compatibility, the availability of tools and benchmark games, and the potential applicability of optimizations to its shaders. However, both OpenGL and Vulkan generally use GLSL shaders, and the core concepts of the graphics pipeline remain the same no matter which API is in use, so the findings here should still be generally applicable across all APIs.

2.4 Game Engines

This section provides a rough overview of the purpose of game engines within typical game software. This background information will help explain why different games using the same engine may have similar shader code despite having drastically different art-styles, and why automated cross-compilation of shaders between different graphics

APIs is a common occurrence[38][39][40].

2.4.1 The Role of an Engine

Game engines are middleware that allows many complicated technical details about different platforms and graphics APIs to be abstracted away from the specifics of individual games[49]. They provide a set of core systems and libraries that handle common tasks such as simulating physics[50][51], playing audio[52], and rendering graphics[13][53] to the screen. The same underlying engine technology may be re-used across numerous different games with drastically different gameplay and art-styles.

Another goal in many engines is cross-platform compatibility. As games evolved from small assembly-language programs for custom arcade-machines, and began to run across a wider range of PCs and consoles, the use of engines became increasingly necessary[54]. If a game has to run on mobile, consoles, and multiple PC operating systems, rewriting the entire rendering system for different graphics APIs would require significant developer effort. To avoid this, game engines provide an internal rendering abstraction layer with a unified interface to multiple graphics APIs[55][56][41]. This enables artists to develop a coherent look for games across all platforms, with only the rendering engineers implementing the engine needing to account for platform-specific aspects.

2.4.2 Game Engine Development

Many larger development studios build their own in-house game engines that they can re-use between different titles they release[57], often with some degree of engine upgrades between each game. Some developers also licence their game engines to other studios, or sell them as the primary product they produce[58][59]. This results in an ecosystem where many smaller companies use the same core rendering technologies from popular commercial game engines, whereas larger companies develop custom engines[60].

Unity

The Unity engine[59] accounts for over 50% of the game engine market share, with 3 billion apps created in Unity downloaded every month[61]. As with many popular commercial engines, it is available for free with several optional upgrade tiers[62].

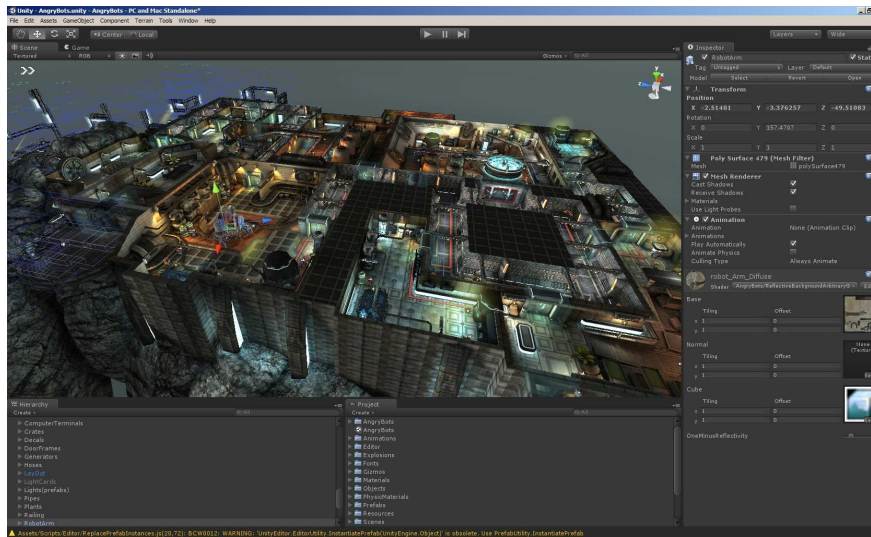


Figure 2.7: The Unity engine's integrated development environment

Alongside the engine compatible with numerous PC, mobile, and console targets, Unity also provides a complete integrated development environment, as shown in [Figure 2.7](#). This contains numerous visual tools for building games by placing 3D objects in levels and using scripting languages to customize their behavior. They also offer an online store for buying and selling pre-made art assets, levels, scripts, or tools[63]. Unity's comprehensive features, robust visual editors, and variety of pre-made content make it simple for developers to create games with minimal programming experience, making it a popular choice among independent developers and larger studios alike[60].

Unity's rendering engine offers many advanced graphical features straight out of the box, and is flexible across a range of 3D and 2D art-styles. On Linux, Unity uses OpenGL, or Vulkan in more recent versions. Its GLSL shaders are automatically cross-compiled from HLSL ones written for the DirectX path in its rendering system[64].

Unreal

The nearest competitor to Unity is the Unreal Engine[58] by Epic Games[60]. Like Unity, Unreal offers extensive visual editors (see [Figure 2.8](#)), an online asset store[65], powerful cross-platform rendering capabilities, and initial free entry for developers[66]. In addition to licensing the Unreal Engine, Epic Games uses it for their own games such as Fortnite, which has over 400 million registered players[67].

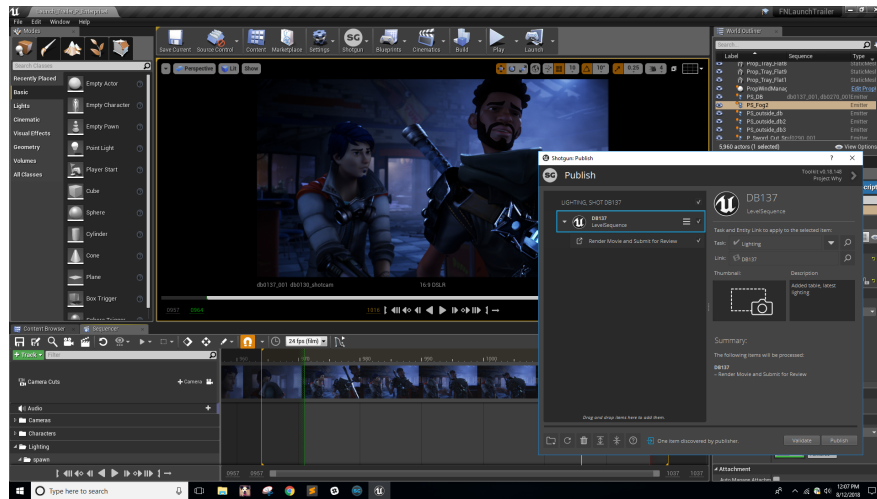


Figure 2.8: Unreal Engine's integrated development environment

Other Engines

Valve's Source engine is used by several popular games, as well as Valve's own first-party series such as Counter Strike, Half Life, and Portal[68]. For 2D games, Game Maker Studio by YoYo games[69], and RPGMaker by Kadokawa[70] are often choices for smaller independent developers. Several open-source game engines are also available [71][72][73], although they are usually less feature-rich than the market leaders Unity and Unreal.

Summary

Game engines are a core part of the graphics ecosystem, and it is uncommon for developers use graphics APIs directly unless they are implementing an engine. Large commercial engines like Unity and Unreal dominate the market, but larger developers also frequently develop their own in-house engines. When selecting different games as benchmarks in Section 5.5 and Section 6.2, it was important to include games from a variety of different engines, while ensuring the most popular engines were represented.

2.5 Programmable Pipeline Overview

This section explains the different stages of the GPU graphics pipeline. It also outlines how data is passed between these stages, and from the CPU to the GPU, which will help explain the different analysis passes performed in subsequent chapters.

2.5.1 Evolution of the Programmable Shader Pipeline

Originally, GPUs were limited to fixed-function data processing, performing only a few hardware-accelerated functions such as vertex transformations, lighting effects, texture filtering, or colour-blending[74]. The special-purpose hardware made these operations fast, but provided limited flexibility for different algorithms and artistic choices.

Despite the limited scope of the early fixed-function GPU pipelines, some games managed to utilize this hardware over multiple different rendering passes² into intermediate buffers to perform complex custom lighting effects[75] (see [Subsection 3.2.3](#) for more details).

Over time, GPUs became increasingly programmable, with newer OpenGL versions introducing developer-written graphics shaders that would execute arbitrary code on the GPU[76]. Programmable vertex and fragment shader stages emerged to allow developers to write arbitrary code to manipulate vertex positions and pixel output colour values, alongside a few remaining hardware-accelerated fixed-function operations such as rasterization and depth-buffer testing. The emergence of programmable shaders is covered in more detail in [Section 3.2](#).

As GPUs became more capable of generalized computation, their design became suitable for solving other non-graphics-related problems, giving rise to general-purpose GPU computing (GPGPU)[77]. Several other developer-programmable shader stages were also added in the form of tessellation[78] and geometry shaders[79]. Compute shaders were introduced to allow general purpose GPU computations to run seamlessly alongside graphics pipelines[80], often running at the start of a game's frame before the graphics pipeline is ready to render. These are common for accelerating physics simulations like cloth[81], fluid[82], smoke, or other particle systems[83]. Recently, ray-tracing using an alternative shader pipeline has also been gaining popularity[84].

Originally, GPUs were designed with separate hardware units for vertex and fragment shaders[85]. However, these were replaced by numerous homogeneous shader/-compute execution cores capable of handling the increasing variety of arbitrary computations and shader types available on GPUs[86]. These shader cores run code in parallel in a series of "warps" where multiple computations occur in lock-step on different input data, and all calculations use a shared set of registers, caches etc on that

²A rendering pass is one full execution of the graphics pipeline. Instead of outputting directly to the screen, pixel colours can be stored as a texture and read from in subsequent iterations of the rendering pipeline i.e. multiple passes.

core[87].

In general, GPU performance is increased when the number of threads in each warp is maximized, as more calculations can occur in parallel. Contention for limited resources such as registers can limit the number of threads per warp, therefore reducing occupancy and causing slow-downs. However, some scenarios may benefit from reduced occupancy, for instance memory-heavy shaders that would quickly overflow the cache and suffer expensive cache misses if too many threads were scheduled at once[88]. These optimization trade-offs are often counter-intuitive, making it difficult to profile and predict the right decisions in advance.

2.5.2 Modern Pipeline Layout

This section describes all the pipeline steps required to transform the data and shaders loaded by the CPU into a fully rendered set of pixels on the GPU. This process includes both the fixed-function steps, and all five potential programmable shader stages. [Figure 2.9](#) shows a simplified diagram of the pipeline featuring only the compulsory vertex and fragment shader stages. Precise implementation details of each stage differ between GPU vendors, but the high-level overview of each stage presented here remains the same across all OpenGL-compatible GPUs.

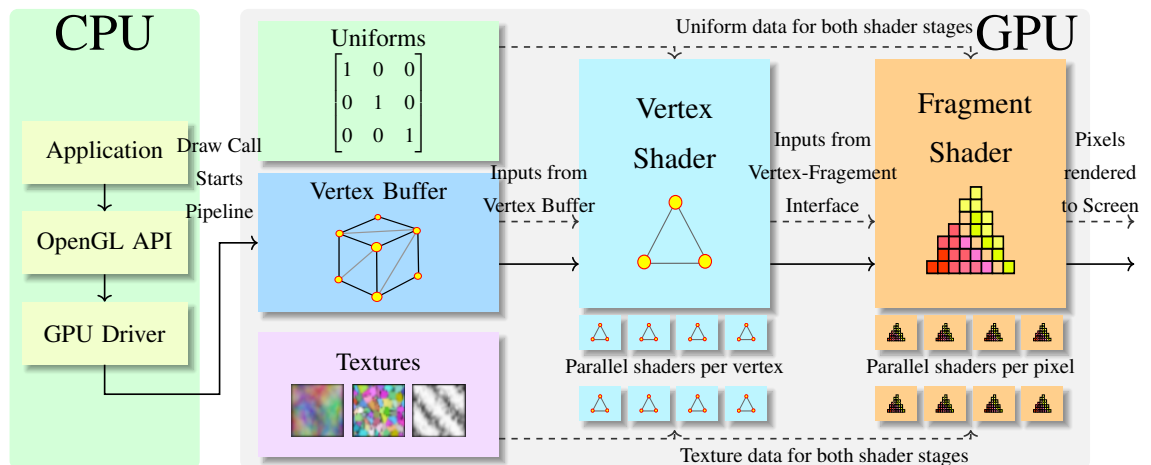


Figure 2.9: Simplified graphics pipeline. The CPU submits draw calls, and the GPU loads triangles from the vertex buffer for vertex shaders to process in parallel. Fragment shaders receive the results, then calculate pixel colours in parallel to render on screen.

2.5.2.1 Buffers and Shaders Set Up

When games first load, shader code must be read from disk and compiled via the GPU driver. It is also possible to cache compiled shader results to speed up subsequent loading times[89]. 3D model data, 2D texture data, and other relevant buffers are then filled in, and flagged to be transferred to the GPU. It is common for much of the shader compilation and data loading to occur when the game is first launched, and subsequent specific art assets to be loaded in batches when the player enters a new level.

Some games also stream assets to the GPU progressively as players advance to avoid lengthy loading screens[90]. If data or shaders required for rendering are not available, the graphics pipeline may stall momentarily. This causes visible stuttering and unresponsiveness, which can be frustrating to players expecting a smooth frame-rate throughout[91]. To avoid this stuttering, it is common for most assets to be loaded in a batch at the start of levels to ensure all data is available on the GPU before rendering[92].

2.5.2.2 Draw Call Submission

Once the initial data is loaded, the core game loop is entered and objects are rendered to the screen in a series of frames[93]. In each frame, individual objects or scenes get submitted to the GPU using draw calls. These graphics API function calls indicate that a set of buffers should be rendered using the current shader program and settings.

Newer APIs like Vulkan provide fine-grained control over when draw calls are sent to the GPU by explicitly recording them into command buffers and then manually submitting them for execution. In OpenGL however, command buffers are implicitly filled by the OpenGL driver, and internal heuristics determine when they are executed[94]. This simplifies initial development, but can make performance unpredictable and hard to tune, resulting in the current trend towards lower level explicit APIs.

Draw calls are often computationally expensive operations, so it is common to batch static models that use the same shader together into a single combined draw-call[95]. Context-switching between different shader programs to draw objects with different materials can also incur overheads, so draw calls are sometimes grouped to render objects using the same shaders together. Switching framebuffer targets, vertex buffer formats, and various other state variables can also incur overheads[96], so the order in which draw calls for different objects is submitted can often heavily impact performance.

If objects in a scene occlude each other, many fragment shader calculations can be omitted. As such, it is often advisable to render objects front to back with respects to the current camera position. However, this must be balanced to avoid context-switching overheads between different shader pipelines too, so ordering draw calls[97], and batching them for efficient dispatch[98] to the GPU are key areas for performance optimizations within many game engines.

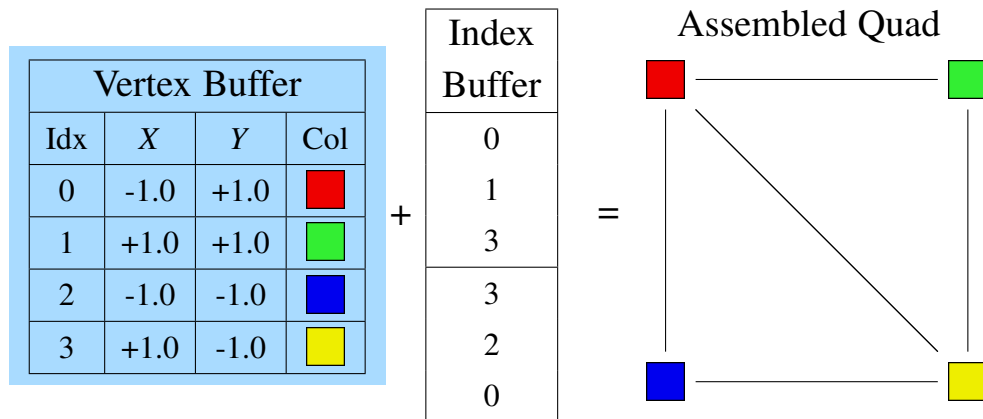


Figure 2.10: A square made up of 2 triangles represented by a vertex buffer storing each point's coordinate and colour data, and an index buffer describing the vertices' order. Triangles are assembled clockwise in the order 0,1,3, then 3,2,0. Note that indices 0 and 3 are shared between triangles, but reuse the same vertex buffer entry.

2.5.2.3 Input Assembly

Data describing the vertices for all the triangles in a 3D model must be assembled from various buffers into the correct format for vertex shaders to process. Triangles are commonly represented using two separate buffers: an index buffer, and a vertex buffer[99]. The vertex buffer holds data associated with each vertex within the model. This per-vertex data usually consists of a 3D position in the model's reference frame, and may include texture UV-coordinates, surface normals, and colour hints for that vertex.

Vertex data is carefully packed and aligned by storing numbers at varying levels of floating point precision, or using normalized integer representations depending on what each value represents[100]. By carefully packing this data, it can be cached and loaded efficiently, and also the overall memory footprint can be minimized while attempting to avoid loss of perceptual quality in the rendered image[101].

The index buffer describes the order in which different vertices should be assembled into triangles to form the overall mesh. In a closed mesh, vertex data is often reused between multiple triangles where they intersect. Separating the vertex and index buffers allows a single integer index to be repeated, rather than storing multiple copies of each vertex's full data entry, as seen in [Figure 2.10](#).

The order in which indices are read in and interpreted as triangles depends on various API settings, and it is often possible to read triangles as a fan, strip, or list, depending on how the index buffer is packed[\[102\]](#). Vertex buffer data can also be ordered to allow it to be read in an efficient order using the index buffer by placing data for nearby vertices close together in memory[\[103\]](#). The input assembly stage reads both the index buffer and vertex buffer, and presents the data to the subsequent vertex shading stage for processing. It retains information about which vertices belong together so that results can be blended across each triangle's surface.

2.5.2.4 Vertex Shader

Vertex shaders take data from each of the triangles' vertices and process them in parallel. The primary purpose of most vertex shaders is to transform the vertex's 3D coordinates from model space to camera space[\[104\]](#). This is usually done by multiplying the position and surface normal by a 4x4 transformation matrix containing a representation of the model's position and orientation within the 3D world[\[105\]](#). This matrix can be updated each frame based on user input, and then multiplied by the static position data within the vertex buffer to simulate the object's motion. This approach means that only a single matrix needs updated each frame, rather than transforming and re-uploading all the position data in the vertex buffer whenever the object moves.

The same principle also applies for more complex skeletal animation techniques. The vertex buffer stores information about which vertices are affected by which virtual "bones". The vertex shader then calculates where each point should be moved based on the position and orientation of the model's virtual skeleton[\[106\]](#)[\[107\]](#). This skeletal animation data can be stored in a separate significantly smaller buffer, thus minimizing the amount of data transferred to the GPU for each frame of animation. Vertex shaders can also offset positions using height or bump maps for effects such as explosions, ocean waves, or terrain (as shown in [Figure 2.5](#)).

After processing each vertex's data, all the vertex shader's outputs are then interpolated across the surface of each triangle. This interpolation also includes perspective corrections to make the output appear more visually correct within the 3D space of

the camera frustum[108]. In addition to values transformed or calculated by the vertex shader (e.g. positions or surface normals), values loaded unaltered from the vertex buffer (e.g. texture UVs or surface colours) may also be submitted for interpolation. This ensures all samples across the triangle's surface in subsequent stages are smoothly blended between the different values at each of the three vertices.

2.5.2.5 Tessellation Shaders

Tessellation[78] is an optional stage in the graphics pipeline, which many smaller games, especially those targeted at lower-end hardware and mobile devices tend to omit. Tessellation is the process of splitting existing triangle geometry into a larger number of smaller triangles, as shown in Figure 2.11. The benefit of this, is that smoother or more detailed surfaces can be generated from a smaller set of explicitly defined triangles. As real geometry is generated, these extra details can show up in silhouettes, and can also be displaced by bump or normal maps.

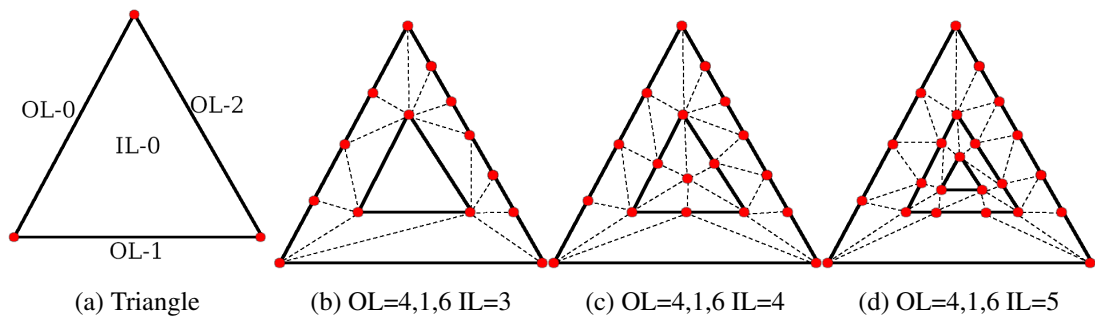


Figure 2.11: A triangle undergoing tessellation with various combinations of outer level (OL) and inner level (IL) edges.

Tessellation shaders run in two parallel shader stages - control, and evaluation[109]. These shaders determine the points at which to split the current triangle. This splitting can be performed efficiently in parallel using special-purpose hardware. Calculations within tessellation shaders often have to be more careful about floating point precision. Different shader invocations with vertex data in different orders must produce identical results to maintain a watertight mesh without miniature cracks appearing between generated triangles[110].

Tessellation shaders are one of the few areas within the graphics pipeline where exact floating point values generated will have significant effects on the outputs. However, they are seldom used within Linux-compatible games, so optimizing them is

beyond the scope of this thesis. Avoiding the tighter constraints of tessellation shaders enables additional unsafe floating point optimizations to be explored, as most other pipeline stages are more lenient about precision.

2.5.2.6 Geometry Shader

Geometry shaders[79] are another optional stage in the graphics pipeline. As with tessellation shaders, they are usually omitted, especially on Linux and mobile devices, so will not be extensively explored within this thesis. A geometry shader's purpose is to generate additional primitives, which may be points, lines, triangles, quads, or polygons. This is useful in scenarios such as turning individual points in a particle system (e.g. smoke, fire, sparks) into billboard quads that can be textured[111]. Other applications can include adding detail to surfaces via decals[112], or generating a cube-map in a single rendering pass by duplicating and projecting geometry onto each face[113]. Despite these various uses, geometry shaders are rarely in practice, and multi-pass algorithms, general-purpose compute shaders, or other stages in the graphics pipeline are often used for these techniques instead.

2.5.2.7 Clipping and Rasterization

After the final geometry's position has been calculated by either the vertex shader or the optional tessellation and geometry shaders, the next step is to determine which parts will be visible on screen. In OpenGL, the clip-space for all visible objects is represented as a cube from -1.0 to +1.0 in each axis[114]. Triangles projected fully outside this clip space can be discarded from further processing. Some triangles can also be eliminated using back-face culling, which removes triangles that are facing away from the screen. Triangles partially within the clip-space cube must be split into smaller sub-primitives at the clipping boundaries before further processing. Any remaining triangles within the clip-space cube are projected into screen-space for rasterization[104].

Rasterization is the process of filling in all visible triangles with pixels[115], as shown in Figure 2.12. Pixels generated by the rasterization hardware are sampled by fragment shaders, which determine their colour using the linearly interpolated vertex shader results at that point on the surface. Both clipping and rasterization can be handled by specialized hardware, and occur automatically as part of the graphics pipeline with no programmable shader intervention.

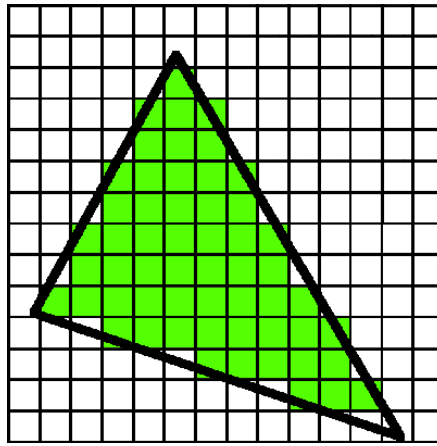


Figure 2.12: Rasterization selects which pixels in each triangle must be filled in.

2.5.2.8 Pre-Fragment Operations

Once their final positions have been decided, some pixels may be eliminated to avoid calculating their final colours entirely, thus speeding up the fragment shading stage. One pixel culling method is to remove those that occur behind pixels calculated by previous fragment shader invocations. To do this, a technique called depth-buffering is commonly used[116] [117]. Here, a buffer records the clip-space Z coordinate for all pixels, and is gradually filled by all draw calls in a frame[118]. This Z-buffer can be checked to see whether pixel values have already been calculated closer to the screen by comparing the current and previous depth values. If the current depth value is closer to screen, the fragment shader can proceed to overwrite that pixel. Otherwise, the colour value is safe to avoid computing entirely, unless the fragment shader itself re-calculates the pixel's depth. The depth buffer is cleared between each frame, and provides an efficient way of avoiding wasteful fragment shader invocations. Another way to eliminate fragment calculations is using stencil buffers[119], which provide a boolean mask describing which areas to render on the screen. Using both depth and stencil buffers where possible can provide significant performance boosts.

2.5.2.9 Fragment Shader

Once the rasterizer has determined the final pixel positions to sample, and discarded all unnecessary ones, fragment shaders run in parallel to emit the final output colour at each position. As with other shader stages, fragment shaders may perform arbitrary calculations. These usually involve lighting calculations and sampling from textures at the UV coordinates passed in from the vertex shader. In multi-pass rendering, the final

output results may not represent colours directly, and can encode other information such as material IDs or surface normals[120]. The final results are outputted into a frame buffer, which may be used as a texture for future rendering passes, or transferred directly to the screen.

2.5.2.10 Post-Fragment Operations and Colour Blending

Once the fragment shader invocations have occurred, the outputted colour values are written into a frame buffer. For opaque objects, new colours may simply overwrite the old values. However, transparent objects require colour values to be read from the frame buffer and blended in a weighted mixture with the newly calculated colours[121]. This may occur as part of the fragment shader, or in a fixed-function step. Once the final colours have been sampled using fragment shaders and blended, then the output buffer can be sent to the screen.

2.6 Transferring Data to the GPU

Now that the programmable graphics pipeline's various stages have been explained, this section will cover the main ways in which data can be passed from the CPU into the GPU's shaders. Some of these mechanisms have already been mentioned, such as textures (see [Subsection 2.2.2](#)), vertex buffers, and index buffers[99] (see [Subsubsection 2.5.2.3](#)). This section introduces the concepts of uniforms[122] and uniform buffers[123], which are the focus for many optimization opportunities used in future chapters, especially [Chapter 6](#). Typical techniques for manipulating this data within the OpenGL API will also be explained.

2.6.1 OpenGL Buffers

The OpenGL API utilizes buffers, which represent an arbitrary block of data and its associated metadata. This includes information such as whether the buffer is read-only, or hints about its size, alignment, usage, or synchronization, which can enable various optimization heuristics within the OpenGL driver[124].

OpenGL is a heavily state-based API, and uses the concept of "binding" a buffer to a specific "slot". Depending on what type of slot the buffer is bound to, the data within it can be reused in different ways. Buffers can be written into using API calls that either discard and replace the entire contents, or overwrite only partial subsections. It is

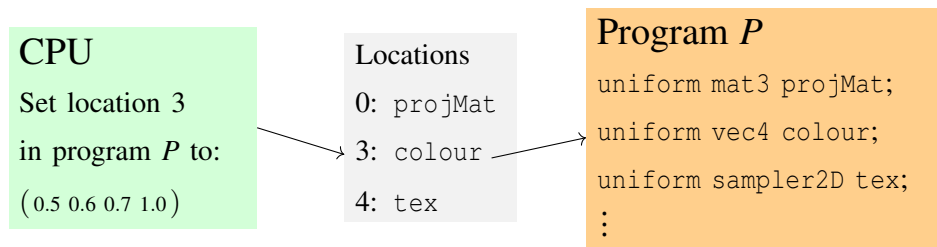


Figure 2.13: Assigning a value to a uniform. The shader compiler determines a “location” for each variable, and a CPU API call assigns values to that location.

also possible to memory-map buffers to regions of the main CPU-accessible memory, and write into them using traditional C++ memory modification methods. Memory-mapped regions can be set up to synchronize in different ways so that changes to the CPU-side memory are reflected into the GPU-side memory implicitly by the driver. On embedded GPUs, the CPU and GPU may even share the same memory region directly[125].

A buffer’s contents can be associated with various different purposes depending on what sort of “slot” it is bound too. These uses include vertex and index buffers, and coloured pixels in texture images[126]. Buffers may also represent regions of memory to be written into by shaders, and eventually read from by the CPU after the contents have been synchronized. Although it is possible to bind a single buffer to multiple different types of targets (e.g. textures, index, or uniform buffers), it is uncommon in practice, and can interfere with the driver’s optimization heuristics. More commonly, buffers are created for a single purpose, and then maintain that purpose throughout their lifetime.

2.6.2 Uniforms

Shaders run in a series of parallel invocations executing the same code on different input data (e.g. different vertices of a model’s triangles, or different pixels of the final image). However, some data may remain the same across all invocations, such as the global transformation matrix determining the model’s position and orientation within the virtual world, or various settings about lights, skeletal positions, or material properties. Data that remains constant throughout all shader invocations in a single draw-call is referred to as “uniform” data[122].

Uniform data is typically frequently updated, and far smaller in size than the per-vertex data stored in the vertex buffer. Some uniforms, such as the camera position

may update once per frame, and some data such as individual model position matrices may be updated once per draw call. As uniforms are constant across all shader invocations in a single pipeline pass, they can be loaded into a small fast cache during execution[127]. However, as they are updated frequently, they may incur data transfer and synchronization costs every time they are updated for a new draw call.

Traditional OpenGL programs represent individual uniform variables as having a single "location" determined by the shader compiler. CPU-side API calls can assign data to each variable using this location as an identifier, as illustrated in Figure 2.13. This fine-grained uniform addressing can help minimize the amount of data transferred if only small subsets of locations require updating. This also enables dead-code elimination to elide unused uniform data by not assigning them locations[128].

If many uniforms need updated at once, however, sending numerous API calls to update small portions of GPU-side data can incur significant overheads. To avoid this, some games use large arrays of floating-point vectors to represent all their uniforms, and then update them in single batch to avoid repeated individual updates.

Uniform data is uniquely associated with a single programmable shader pipeline. Each pipeline must store all uniform data associated with it. This may be inefficient in cases where data could be re-used between multiple shader pipelines, for instance the camera position, or various scene-level lighting settings. Uniform buffers can provide a solution to this problem, as described below.

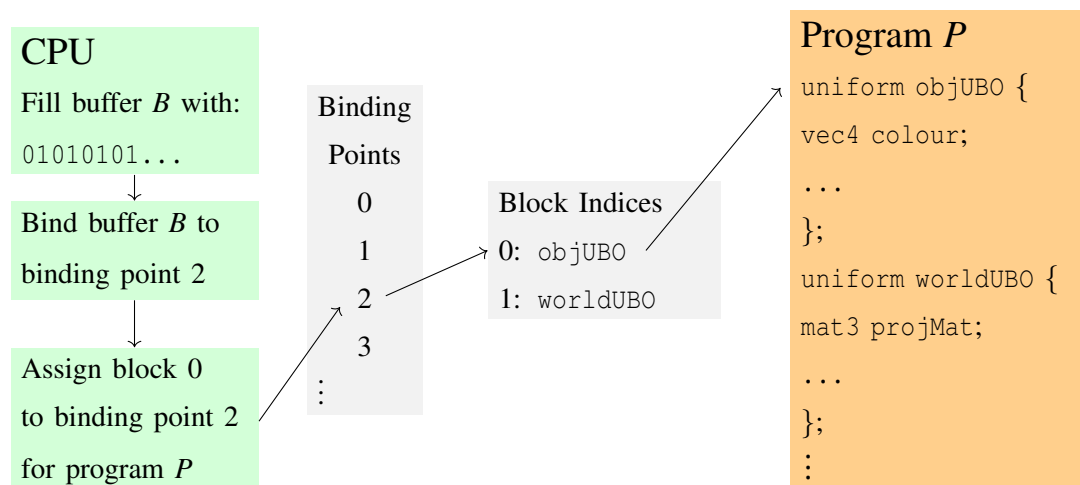


Figure 2.14: Binding a uniform buffer object (UBO). Data is written to a buffer, and bound to a global CPU-side UBO "binding point". The shader defines a block of variables, that is assigned an index by the compiler. The CPU matches per-program block indices with global UBO binding points, so the GPU can read the bound buffer's data.

2.6.3 Uniform Buffers

Uniform buffer objects (UBOs)[123] are a more modern addition to OpenGL. They can store larger amounts of uniform data, and improve data sharing between shader pipelines. UBOs can be filled and updated using the general buffer techniques[124] described in [Subsection 2.6.1](#), and can then be bound to different compiler-determined slots within a shader. UBO data must be packed and padded in such a way that the shader code reading it can interpret it as variables. These variables must correspond to the interface in the shader for that particular UBO binding point[129]. The shader code indexes into the buffer using structured variable names, and the GPU must load data from the memory region corresponding to the buffer bound at that UBO index, as shown in [Figure 2.14](#).

Shader pipelines may have multiple UBOs bound to them. For instance, one UBO might represent global per-frame state such as camera positions and scene lighting, and may be bound to many pipelines. Another smaller buffer might also be attached containing per-object data unique to a single pipeline. Instead of loading new uniform data between each draw call, UBO binding indices can be updated to point to different buffers already loaded on the GPU. This allows data transfers to be efficiently batched, minimizing how often they occur. The improved data sharing capabilities of UBOs can also aid caching behaviours and minimize the overall memory footprint.

UBOs provide data to shaders via the buffer update API[124], rather than the location-based per-pipeline uniform variables[122] described above in [Subsection 2.6.2](#). Despite this, the general concepts of UBOs and traditional uniforms remain largely similar despite their different implementations.

2.6.4 Shader Storage Buffers

Shader storage buffer objects (SSBOs)[130] are another method for transferring data between the CPU and GPU, offering larger buffer sizes than UBOs. SSBOs can also be written into by shaders, unlike UBOs, which are read-only. These properties make SSBOs more flexible but harder to efficiently cache than the more specialized UBOs. They are bound to shader pipelines and accessed with similar method to UBOs described in [Subsection 2.6.3](#). SSBOs are used less commonly in games than UBOs, and as they are not read-only, the data analysis within this thesis focuses entirely on UBOs and traditional uniforms.

2.6.5 Textures

Texture data can be filled in using the OpenGL buffer interface[124] described in [Subsection 2.6.1](#). However, textures must also provide additional metadata such as pixel colour encoding, compression, filtering, and tiling settings[131]. Texture data is mostly static and transferred to the GPU in advance. A small number of dynamically updated textures may also be used for physics or visual effects such as wind, footsteps, snowfall etc. [132]. These may be either re-uploaded to the GPU each frame, or written into by compute shaders or the graphics pipeline.

To determine which texture buffer to read from, shaders define a set of "sampler" variables[133]. Samplers are integer uniforms representing the index of a texture "slot". Shaders provide this sampler ID to `texture` functions, which read colours from the buffer bound to that slot. Calls to `texture` also require UV coordinates to determine where to sample within the texture buffer, as explained in [Subsection 2.2.2](#).

On the CPU-side, buffers are bound to texture slots, as illustrated in [Figure 2.15](#). Texture slots are global OpenGL state accessible to all programs. The texture slot's index is assigned to sampler uniforms within the shaders using the method described in [Subsection 2.6.2](#). The buffer bound to the corresponding texture slot when a draw call is submitted will then be read from by the shader.

As explained in [Subsubsection 2.5.2.2](#), draw calls may be stored in command buffers and dispatched to the GPU in batches. To allow for this, a snapshot of the globally bound texture buffers may need to be recorded alongside draw calls to ensure the OpenGL execution model appears as a single deterministic thread.

Although samplers[133] are uniform variables, they cannot be stored within UBOs. Instead, they must use the traditional per-shader-pipeline model of setting uniform values[122] described in [Subsection 2.6.2](#). Some games, including some benchmarks used in [Chapter 6](#), use UBOs for all uniform values except texture samplers. It is common for sampler ids to be set once per shader pipeline. Different texture buffers can then be bound to the same slot ids for different draw calls.

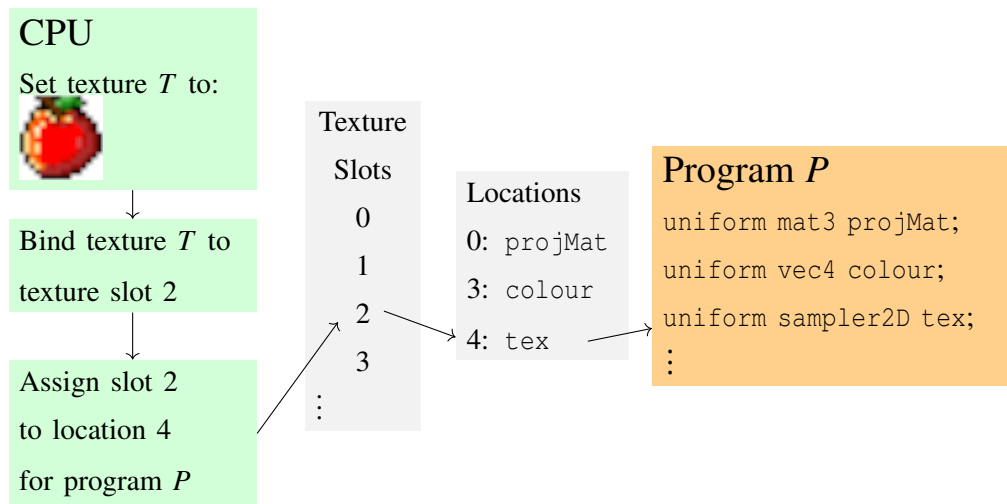


Figure 2.15: Assigning a texture sampler as a uniform. A texture is stored in a buffer and bound to a global texture slot. A location-based sampler uniform variable (see Figure 2.13) determines which texture slot to read from when a draw call is issued. Sampler IDs cannot be stored in UBOs, and must use traditional uniforms.

2.7 Summary

This chapter introduced the concept of the 3D graphics pipeline and the ecosystem around it. Below is a summary of key topics that will be used in subsequent chapters.

Standardized graphics APIs enable cross-compatibility with hardware from different GPU vendors. Although OpenGL[4] is used throughout this thesis, most concepts are applicable to all modern graphics APIs. These details are often abstracted away by game engines, which can be re-used for many different games.

The graphics pipeline transforms triangle-based 3D models into a set of coloured pixels. Games are rendered as a series of frames, each of which consists of multiple draw calls. When a game first loads, CPU-side OpenGL calls initialize state and transfer data to the GPU. Draw calls trigger a pipeline of small GPU-based shader programs (written in GLSL[36]) to execute on the current set of buffers.

The most common shaders in the graphics pipeline are vertex and fragment shaders. Vertex shaders read the 3D model's triangle from the vertex buffer. They execute in parallel once per vertex and calculate their position on the screen. Fragment shaders run in parallel roughly once every pixel in the final output image, and compute their colour. 2D texture images can be sampled to aid in computing the colours of the 3D model's surface.

Uniform variables[122] are data that remains constant across all parallel invoca-

tions of a shader, e.g the model's position or lighting effects. Uniforms are frequently updated between game frames, or between different draw calls. Uniform buffer objects (UBOs)[\[123\]](#) can upload uniform data in batches, as opposed to the traditional per-variable approach to uniforms. Samplers are a special type of uniform representing the index of a texture that can be read from.

Chapter 3

Related Work

3.1 Introduction

The aim of this chapter is to contextualize this thesis by providing overviews of several research fields adjacent to this work. It begins with a history of programmable shaders in [Section 3.2](#), which explains the evolution of vertex/fragment shader pipeline at the core of this research. [Section 3.3](#) then explores various techniques for simplifying shader code, most of which require trading visual quality for improved performance. In [Section 3.4](#), a class of optimizations is introduced which utilize knowledge of the values that variables will take, typically through some form of run-time analysis. [Section 3.5](#) describes the history of optimizing games for low power usage, and the workload characterization approaches involved. Finally, [Section 3.6](#) covers various tools and approaches that can be used for debugging and profiling GPU programs.

3.2 Evolution of Programmable Shaders

The modern graphics pipeline and its programmable shader stages described in [Chapter 2](#) has been remained largely unchanged throughout the last decade, but emerged from a large body of work experimenting with different conceptualizations of how rendering systems should work, and how programmers should be able to modify them.

3.2.1 Early Shader Conceptualization

Whitted et al. developed a scanline-based rasterization system that determined the colours to render various surfaces using small "shader" programs implementing dif-

ferent lighting algorithms [134] [135]. Cook suggested the concept of "shade trees" [136], which could describe shading, lighting, texturing, and atmospheric effects in a node-based fashion that could easily be combined and re-used to tweak the appearances of different objects and scenes. Abram et al. built a GUI-based implementation of Cook's shader trees to show the utility of being able to create libraries of combined shader sub-trees for various graphical effects [137].

In Perlin's image synthesizer paper, he suggests a language for a "pixel stream editor", which can be applied over multiple passes to alter the appearance of the output image, and create complex realistic effects using procedural noise functions to mimic marble, water, clouds etc. [138]. These can be seen as somewhat analogous to modern fragment shaders.

In 1990, Hanrahan et al. built on many of Cook and Perlin's ideas in the creation of the Renderman shading language [139]. This was implemented as part of a CPU-based ray-tracing framework, and allowed great flexibility in lighting and shading effects.

3.2.2 Early Rendering Hardware

To this point, much of this work on shading systems was being performed on standard CPUs, and was not able to achieve speeds capable of real-time rendering. Other researchers were exploring specialized hardware architectures to exploit the inherent parallelism pervasive in many graphics problems, such as the GSP-NVS [140], the Pixel-planes 4 [141] [142] and 5 [143][144], and the Pixelflow system [145]. These systems were capable of performing various fixed-function shading operations such as Gourard [146] or Phong shading[15] [147], as well as various levels of texture-mapping capabilities.

Despite the impressive rendering speeds of the hardware-accelerated rendering architectures above, several researchers noted that there would be great utility in allowing programmable shaders to be part of these pipelines. Rhoades et al. suggested a low-level language to support procedural texturing[148] within the Pixel-planes 5 system. Lastra et al. [149] suggest a method for incorporating a modified version of the high-level Renderman shading into the Pixelflow architecture [145], enabling much more flexibility in the types of custom shading algorithms available.

By 1997, the Pixelflow system was finally completed [150], and they chose to use a variant of the OpenGL API[151] as its interface. OpenGL was used by games and other real-time graphics systems at the time, but had no support for programmable

shaders, instead specifying a fixed shading and lighting model. Olano et al. describe how they implemented support for fully user-programmable shading [152] in the modified Renderman shading language suggested by Lastra et al., and how the OpenGL API could be extended to support programmable shader stages too[153]. This system demonstrated that complex real-time rendering systems with fully programmable shaders were able to run at 30FPS by using highly parallel hardware.

3.2.3 Multi-Pass Shading

An alternative approach to programmable shading in OpenGL was to use the existing OpenGL API function calls as a target for a compiler, rather than adding new features via an extension. Peercy et al. demonstrated this technique [154], treating the OpenGL as a general SIMD computer, and compiling both the Renderman shading language, and their own high-level Interactive Shading Language (ISL) into a series of OpenGL API calls over multiple rendering passes. This research built upon prior works demonstrating how complex rendering results could be approximated using multiple passes[155] [156] through the limited fixed-function graphics pipeline in real-time, allowing them to take advantage of more common graphics acceleration hardware. Similar multi-pass shading techniques in OpenGL were demonstrated by the scripting language used in Quake 3 by id Software as well[75].

3.2.4 Programmable Shaders on GPUs

In 2001, NVIDIA presented a paper on the programmable vertex units they had added to their GeForce 3 GPUs[157]. These early programmable vertex units used a low level assembly-like language with many constraints, such as the inability to perform branching instructions, but provided an initial basis for bridging the gap between the old fixed-function shading process, and newer more flexible programmable shading. Guided by this work from NVIDIA, low-level GPU programability for both vertex and fragment shaders[158] was added to Microsoft's DirectX 8 API, and to OpenGL via the use of extensions.

Proudfoot et al. developed a higher level programming language based on the Renderman shading language[159] (later referred to as the Stanford Real Time Shading Language - RTSL), which allowed shaders to be written at a much higher level that could then be compiled down[160] into the more assembly-like language available at the time.

Mark et al. developed a C-like language called Cg[161] which aimed to be cross-compatible between OpenGL and DirectX, and add features such as branching that were emerging in newer DirectX 9 compatible hardware. Cg became the basis of HLSL, the High Level Shading Language used for DirectX 9.0, and OpenGL developed their own high-level language known as GLSL when it transitioned to OpenGL 2.0.

More detailed information about the state of programmable shading in this era can be found in Ecker's paper on "Programmable Graphics Pipeline Architecture"[162] or in the course-notes on "Real-time shading" from SIGGRAPH 2004 [163].

3.2.5 GPGPU Programming Emerges

Since this period, the high-level structure of hardware-accelerated programmable shader pipelines has not changed drastically. The programmable vertex and fragment shader remain at the core of most modern graphics pipelines, and the hardware running these stages has become more and more flexible.

Researchers took note of the increasingly flexible hardware-accelerated parallel programming model designed for shaders, and adapted it for arbitrary data-parallel computations via systems such as Scout[164], Brook[77], CGiS[165][166], Glift[167], and Accelerator[168]. This body of work formed the basis for subsequent popular general-purpose GPU computing (GPGPU) APIs such as CUDA [169] [170] and OpenCL [171] [172].

With the increasing flexibility of programmable shaders, and the rising interest in GPGPU applications, GPU architectures began to transition away from separate vertex and fragment shader hardware units in favour of a unified shader architecture[86][85]. In a unified architecture, the same generic shader core is used for both vertex and fragment calculations (or GPGPU code), meaning more unified cores can be packed into the same area that was previously split between vertex and fragment cores, allowing for increased parallelism.

3.2.6 New Shader Types

Unified shader architectures also allowed additional stages to be added to the graphics pipeline, such as tessellation and geometry shaders as described in Section 2.5. Graphics APIs also introduced aspects of GPGPU programability via compute shaders, which operated outside the traditional graphics pipeline, but could share data with it.

This allowed certain elements of games such as physics or particle systems to be offloaded from the CPU to the GPU. Recently, NVIDIA have also proposed an alternative to the traditional vertex-fragment shader model using mesh shaders[173], which provides similar flexibility to compute shaders, but have the ability to output triangles directly to the rasterization hardware.

There has also been significant interest in hardware-accelerated raytracing pipelines, such as NVIDIA's RTX[174] introduced in their Turing architecture[175], and AMD's RDNA2 architecture[176]. Raytracing support is being added to both the DirectX 12 API[177] and Vulkan[178], replacing the vertex/fragment shader pipeline with ray generation, intersection, any-hit, closest hit, miss, and callable shaders.

This brings the story of programmable shaders full-circle. Shaders were initially developed in the 80s to add flexibility to offline raytracing systems, but now decades later, main-stream consumer-level graphics hardware has reached the point that raytracing shaders can run in real-time, allowing interactive computer games access to global illumination techniques previously available only in offline renderers for films.

Currently, hardware-accelerated raytracing is only available in cutting edge higher-end GPUs, and few games outside the high-budget AAA releases from larger studios are taking advantage of it. As such, it is unlikely that the programmable vertex/fragment pipeline, that has survived for almost two decades now, will become obsolete any time soon. Even if raytracing shaders become more commonplace, it is likely that they will only form a subset of the rendering system used, as traditional vertex/fragment shaders are better suited to many aspects of rendering.

This thesis focuses entirely on games with programmable vertex and fragment shaders, which form the vast majority of recent games. The techniques applied here are likely to be applicable to other types of shaders too, but the performance characteristics of other pipelines such as mesh shaders or raytracing are likely to be somewhat different, and could be an interesting angle for future work if these novel pipelines become more prevalent in the future.

3.3 Shader Simplification

Graphics is an interesting domain, as the outputs do not need to be mathematically accurate as long as the results appear passably similar to the human eye. As a result, various different techniques for radically simplifying, optimizing, or approximating shader programs have been explored. [Chapter 4](#) examines the effects of several

unsafe floating-point arithmetic reassociation techniques[9], which usually result in visually imperceptible differences. This thesis's second main focus is shader specialization, a concept which has been explored by researchers since before the modern programmable GPU graphics pipeline came into being[179]. This section considers work on even more drastic simplification techniques for shader code, which trade visual fidelity and correctness for increased execution speed.

3.3.1 Level of Detail

In 3D graphics applications, objects far away from the viewer's camera appear much smaller on screen, so subtle changes to the actual details of these objects are impossible to notice. This allows for a performance vs accuracy trade-off in the "Level of Detail" (LoD) the object is displayed in. It is standard practice within the games industry to use radically simplified 3D models with far fewer triangles to represent far away objects[180][181], and also to use lower resolution textures to apply to their surfaces[182]. There are often several transitional levels of detail that can be transitioned between as an object gets closer to the camera, and the flaws in the less detailed version become more noticeable.

3.3.2 Shader Level of Detail

Researchers have explored many automated shader simplification systems designed to generate different variants of shaders for different levels of detail. In early work, Olano et al. propose a system for generating different shade LoDs by splitting code into blocks, and simplifying the code by omitting, mathematically approximating, or substituting some blocks for simpler alternatives of known lighting calculations. They also suggest methods of removing texture look-ups, or combining multiple look-ups into a single one from a combined texture, as part of an automated shader LoD generation system[183]. Simmons et al. suggest techniques for smoothly transitioning between different shader LoDs on a per-pixel basis, and provide a prototype implementation in a very early form of OpenGL programmable shaders[184]. The aim of this per-pixel smooth transition is to avoid jarring pop-in artefacts when the object gets closer to the viewer and suddenly switches to a new LoD.

Pellacini suggests generating shader LoDs by iteratively applying simplification rewrite rules to an abstract syntax tree[185] from the high-level Cg shading language[161]. The correct simplification to choose at each stage is selected by measuring the visual

error that each rule-based substitution causes on a test image. Sitthi-Amorn et al. extended these simplification ideas by using genetic programming and GPU-accelerated testing to explore a wider range of possible shader code transformations[186]. More recently, iteratively applied re-write rules have also been explored for simplifying physically-based material shaders written in high-level declarative languages[187].

Cho et al. [188] and Piao et al.[189], used tracing tools to examine the effect of altering various OpenGL settings on mobile games. They primarily focus on altering OpenGL parameters, such as render buffer precision and texture filtering modes, but also experiment with shader simplifications, such as replacing occurrences of x^y with either x^1 or x^0 . They then explore the effects of these simplifications in terms of a performance-accuracy trade-off with varying levels of detail.

3.3.3 Surface Signal Approximation

Wang et al. explored a different approach to shader simplification that went beyond the scope of prior simplifications on single fragment shaders[190]. They moved many per-pixel calculations to prior stages in the graphics pipeline, and used surface signal approximation to represent fragment shader calculations using high-order polynomial basis functions, which are sampled by smaller patches generated by tessellation shaders, rather than on the previous per-pixel basis. These approximations can be pre-computed and cached to avoid run-time calculations, which is similar to other work on caching of partial graphics calculations, and re-projecting them in subsequent frames to avoid repeatedly recalculating all the results[191] [192].

3.3.4 Altering Computation Rates

He et al. build upon prior simplification work by examining ideas around calculation rate reduction[193]. This essentially means moving certain calculations to prior stages in the graphics pipeline, such as moving per-pixel calculations into the vertex shader instead, as there are typically fewer vertices than pixels. They also suggest moving code from the GPU to the CPU, so that it happens on a per-draw call, or a per-frame basis, rather than on more granular levels. By using approximate common sub-expression elimination, and extensive error caching and estimation techniques, they are able to generate simplified shaders rapidly, along with transitional shaders blending between different LoDs to avoid pop-in artefacts.

Later, He et al. introduce Spire[194], a system for writing and compiling shaders

designed for flexibly moving calculations between different rates, and experimenting with different optimizations and performance accuracy trade-offs in shaders, either manually or via iterative compilation. Yuan et al. take many of the ideas around shader simplification, and introduce heuristics such as clustering similar shaders to reduce the time required traversing the search-space of possible shader simplifications to the point that it can occur as part of a real-time rendering application, rather than as a separate offline phase[195].

The main ideas from the shader simplification literature explored within this thesis are those focused around code motion between different stages of the pipeline to reduce their computation rate. [Chapter 5](#) shows that some of these rate-reduction optimizations are possible within pre-existing shaders, even without having to simplify them, and that constant-folding using data from run-time traces enables even more of the code to be moved between different pipeline stages, or from the CPU to the GPU too.

3.4 Value-based Optimizations

One core idea this thesis seeks to explore is the use of values known at run-time to improve the speed of graphics shaders. Constant-folding using static values has been a standard compiler optimization for decades (see [Subsection 3.4.1](#)), but the use of data known only at run-time has been the focus of many research areas. Hardware-based branch prediction, used to determine whether control-flow branches will be taken at run-time, is one such research area (see [Subsection 3.4.2](#)). The use of hardware mechanisms to predict values loaded from memory is also explored in [Subsection 3.4.3](#).

Software-based techniques for exploiting run-time values have also been heavily researched. These may involve either the use of just-in-time (JIT) compilers to dynamically generate machine code during run-time, or dynamic binary translation tools to modify pre-compiled code. [Subsection 3.4.4](#) explores these fields, and explains the impact the LLVM compiler framework had on these areas. Another compiler-based approach is to use value profiling, as explained in [Subsection 3.4.5](#), to help optimize code during the initial compilation phase by building in fast-paths for common values. Finally, [Subsection 3.4.6](#) covers recent applications of these value-based optimization techniques to GPU programs.

3.4.1 Constant Propagation and Folding

The ideas of constant-propagation and constant-folding using variables known at compile-time have been well-known for over 50 years of compiler optimization research [196] [197] [198]. These remain standard techniques for any optimizing compiler [199] [200]. Further work extended the scope of constant-propagation from a local per-function optimization to a global inter-procedural operation [201][202][203][204], with further extensions to allow constant-propagation within parallel programs [205][206]. Although the shaders explored within this thesis are executed in parallel, they are written as small single-function programs representing the operation of a single thread, so local intra-function constant-propagation and folding are sufficient for this use-case.

3.4.2 Branch Prediction

Constant-propagation and folding require data to be constant at compile-time, whereas many other techniques seek to optimize based on data values known at run-time. One of the most well-known applications of run-time value prediction is the use of hardware branch prediction. In 1972, Riseman et al. noted that instruction-level parallelism was severely hampered by the presence of conditional branch instructions [207]. With perfect scheduling knowledge for their program, and an infinitely deep execution stack, a parallelism ratio of 51:1 was possible. However, if restricted to bypass no more than 2 conditional branches, this was ratio reduced to 4:1.

Some of the simplest forms of branch prediction used in early machines relied on statically made prediction policies. In the IBM System 360 [208], conditional branches were assumed to always fall through, unless a specific "loop mode" was established by a short backwards conditional branch. The IBM System 370 [209] speculatively executed instructions used opcode-based branch prediction, noting that branches with some opcodes tended to be taken more than others. Other machines used dynamic history-based prediction techniques, such as the MU5 [210], which used an associative memory to record whether the next instruction word to fetch was in or out of sequence. In 1981, Smith summarised and evaluated several such branch prediction techniques [211]. He also suggested using hashed addresses in random access memory to avoid the need for associative memories, using small counts of the number of times branches were taken, rather than single-bit histories, and using hierarchies of different branch prediction techniques to improve the overall accuracy.

Since these early implementations, there have been decades of advancement in

branch prediction strategies[212]. Instead of simply recording individual branches' local histories, many techniques seek to take advantage of correlation between branches [213][214], such as two-level[215] and perceptron-based[216] branch-predictors. Chen et al. [217] likened the branch prediction problem to partial pattern matching (PPM) techniques used in data compression[218], leading to several PPM-like branch predictors [219][220]. TAGE[221] is perhaps the most well-known PPM-like branch predictor, and uses geometric history length[222] to exploit both recent branch correlations, and those that occurred far earlier in the program's history. Despite being one of the most accurate standalone branch-prediction strategies[223], TAGE can be improved even further by adding side-predictors for difficult to predict scenarios such as loop termination [224] and branches with a small statistical bias not correlated with its history path [225]. Hybrid branch predictors can also improve accuracy by using multiple solo branch predictors such as TAGE, and combining their results using strategies such as selection[226], fusion[227], or prophet-critic models[228]. With the recent Meltdown[229] and Spectre[230] exploits, branch prediction research has had to cover not only improved accuracy, energy efficiency, and area on chip, but also aim to minimize security vulnerabilities, so continues to be an active area of research.

3.4.3 Value Prediction

In addition to predicting whether a program's branches are taken or not, it is also possible to use prediction techniques for other values. Noting that programs frequently recalculate the same values repeatedly, Harbison proposed a Tree Machine[231] architecture using a value cache to perform common-sub-expression elimination[199] at runtime using hardware, rather than as part of an offline compiler optimization. Richardson built upon these ideas by introducing the concept of trivial calculations, and using a results cache[232]. Lipasti et al. introduce the concept of value locality[233] - the probability that calculations will output the same values during repeated executions. They use this property to speculatively execute load instructions, thereby exceeding the dataflow limit[234] by avoiding pipeline flushes and lengthy waits for memory load instructions to complete whenever the value is mispredicted. Gabbay et al. [235][236] extended Lipasti's "last-value" predictors by adding an additional stride field to their prediction tables to improve their accuracy for certain cases.

In their work on content-aware register files, Gonzales et al.[237] identify the concept of "partial value locality", where some portions of a value's underlying binary

representation are the same even if other portions differ. The work on load value approximation by San Miguel et al.[238] introduces the concept of "approximate value locality", where values are considered similar if they are arithmetically close, even if their binary representations differ drastically. These relaxed concepts of value locality offer new avenues for optimizations based on value prediction and value profiling (see Subsection 3.4.5).

Since this early value prediction work, many other applications of value prediction have been explored, including for GPUs [239] [240] [241] [242][243]. Mittal provides an extensive overview of these advances in value prediction technology[244], but notes that most techniques are experimental and have only been evaluated using simulators, with very few achieving physical hardware implementations to explore real-world results.

3.4.4 Run-time Specialization and JIT Compilation

Just-in-time (JIT) compilation[245] is another field which takes advantage of run-time state for program optimization. One of the earliest implementations of a JIT compiler was in Hansen's 1974 PhD research[246], where he experimented with an incremental approach to program optimization for FORTRAN[247]. Code segments were dynamically recompiled with increasing levels of optimizations whenever each section was executed, enabling frequently used segments of code to become highly optimized without spending unnecessary effort on portions which were only executed once. A similar ethos is still used in modern JIT compilers.

3.4.4.1 Binary Size Reduction

Another early use-case for JIT compilation was to reduce file-size and memory footprints of executables by mixing pre-compiled and interpretable code[248][249]. The interpretable sections could be compiled at run-time, and then simply thrown away after execution[250]. Franz et al. suggested the use of "slim binaries"[251] for the Oberon[252] language, where compressed abstract syntax trees of programs were distributed instead of larger target-specific pre-compiled binary files. These slim binaries were compiled when the program was loaded, and could undergo continuous optimization throughout the the program's lifetime as its run-time state changed[253][254][255].

3.4.4.2 JIT for Dynamic Languages

Early dynamically-typed languages such as APL[256][257], Smalltalk[258][259], and Self[260][261][262] benefited from JIT compilation to improve their performance, as their dynamic type systems meant many static optimizations were not possible. Using JIT compilation to generate optimized native code for "hot" portions of the code instead of running each line through an interpreter or virtual machine (VM) can speed up execution in many scenarios. This is also true for more modern languages using interpreters or VMs such as Java[263][264][265], Python[266][267], and Javascript[268][269][270].

3.4.4.3 Run-time Code Specialization for Compiled Languages

Although JIT compilation is often used to mitigate slow-downs in dynamically typed and interpreted languages, run-time specialization has also been explored for strongly-typed statically compiled languages such as C. In the Synthesis Kernel[271], Massalin et al. used run-time code specialization to improve performance of many aspects of the Unix kernel[272]. Pu et al. later extended these specialization ideas to a real-world filesystem[273] using various partial-evaluation techniques[274].

Keppel et al.[275][276] showed how value-specific optimization (VSO) could be used to speed up programs which contained loops over data that seldom changed, such as the `bitblt` function used to send bitmaps to the screen as a series of scanlines in early rendering systems[277][278]. Anderson [279] developed the C-Mix compiler for partially evaluating C code, and showed its applicability to ray-traced rendering applications[280]. Glück et al. built on earlier work on partial evaluation for Fortran[281] and demonstrated its benefits for various mathematical algorithms such as fast Fourier transforms and solving partial differential equations[282][283].

Consel et al. [284][285][286] built upon these previous approaches, as well as Knoblock et al.'s work on data specialization[287], and aimed to develop a more generalized system for automatically generating templates for specializable portions of the C code. This culminated in the Tempo[288][289] system, which was built on top of the popular gcc compiler[290]. Tempo was designed for optimizing systems applications such as Linux signal-handling, the Berkeley Packet filter[291], and networking systems such as the Sun remote procedure call (RPC) protocol [292].

Tempo was designed to automatically generate specialized programs from raw C code, unlike several other projects operating using custom annotation formats or lan-

guage dialects. ‘C [293] (pronounced Tick C) is a superset of C with additional operators for manually specifying which portions of code should be generated at run time. Its custom compiler, tcc[294] was built on the lcc compiler[295][296], which was designed by AT&T Bell Laboratories and Princeton University to be a C compiler with a shared front-end able to be retargeted to multiple different target-specific back-ends. DyC[297] [298] also used annotations to supplement C code with information about dynamically compilable sections. It was implemented using the Multiflow compiler[299], which aimed to use trace scheduling to maximize instruction-level parallelism in C and Fortran code. The Fabius compiler[300][301] used similar techniques but applied them to a subset of standard ML[302], which demonstrated the applicability of these approaches to functional languages.

3.4.4.4 The Dawn of LLVM

In 2004, Lattner and Adve’s paper titled “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation” drastically altered the landscape of compiler research. Born from Lattner’s Masters[303] and PhD work[304], the LLVM project offered a powerful machine-independent SSA-based intermediate representation (IR) well-suited for implementing many optimizations. Its highly modular design enabled the same optimizations to be used across different language front-ends and generate code for many target-specific back-ends. Its promise of “Lifelong Program Analysis & Transformation” makes it suitable for optimizations during compilation, linking, installation, run-time, and during idle-time between program executions. Much of modern compiler research uses LLVM as a starting point, instead of the wide array of custom research compilers present in earlier works. LLVM is also heavily used in many areas of industry.

Some recent work on using LLVM for run-time value-based specialization is BinOpt by Engelke et al.[305], which translates portions pre-compiled binary executables into LLVM IR to specialize on the fly. BinOpt uses the Rellume[306] library to rapidly lift x86-64 directly into LLVM IR. It improves on their prior work on DBrew[307] and DBrew-LLVM[308] by removing many of the instruction coverage limitations caused by indirect branching. Numerous other JIT-based techniques for C and C++ based on LLVM have also been developed such as Cling[309], LambdaJIT[310], Kahuna [311][312], atJIT[313], and ClangJIT [314].

3.4.5 Value Profiling

An area of research that complements both the value prediction (see [Subsection 3.4.3](#)), and JIT compilation (see [Subsection 3.4.4](#)) is the field of value profiling. In Keppel's thesis on run-time code generation[315], he notes that values which seldom change throughout a program's execution make for good "candidate variables" for value-specific optimizations. Autrey et al.[316] refer to these as "glacial variables", and propose a static analysis pass based on the ideas of staging transformations [317] to determine the "glacialness" of variables. Calder noted that value profiling was useful for both hardware value prediction and run-time code generation, and proposed a convergent run-time profiling technique to top N values of variables to detect "semi-invariant" values[318][319]. Gabbay et al.[320] suggest using value profiling results from simulated executions to allow the compiler to augment instructions with information about whether "last-value" or "stride"-based hardware value predictors are most suitable for each variable.

Muth et al. generalize the approach from profiling value to profiling whole expressions, and show that using profile-guided specialization at link-time can provide significant speed-ups[321]. Chung et al. showed that similar ideas can also be used to improve a program's energy efficiency[322]. Watterson et al. [323] determined that the $\sim 30\times$ profiling overhead of Calder's initial value profiling approach[319] could be reduced by an order of magnitude using goal-directed value profiling. By estimating which variables would be likely to generate profitable optimizations if their values were "semi-invariant", and ignoring all others, the number of variables requiring instrumentation could be drastically reduced. Moseley et al. further reduce the profiling overhead of value prediction by periodically forking a shadow process to be instrumented in parallel, rather than injecting instrumentation into the main executing process [324].

Khan et al. demonstrate a technique for deep value profiling [325] allows for program specialization without drastic binary size increases. Oh et al. use a variant of value profiling to aggressively specialize input-pased loops[326]. Henry et al. [327] implemented a method for directly instrumenting pre-compiled executables with value-profiling using the MADRAS binary patching tool[328] [329] as part of the MAQAO suite. REDSPY[330] is another tool developed for value profiling, which expands value locality to include approximately equal floating-point numbers, enabling additional optimizations to be made for HPC applications where minor accuracy trade-offs

are acceptable.

3.4.6 Value-Based Optimizations for GPUs

Building on the concepts of "approximate" and "partial" value locality developed for value prediction (see [Subsection 3.4.3](#)), Wong et al. noted that the values computed by different threads in a warp are often approximately equal[331]. This approximate intra-warp value locality can be exploited to save energy by executing a single representative thread, rather than calculating exact results for all threads of the full warp. Collange et al.[332] also utilized intra-warp value locality, noting that values in CUDA programs were frequently the same for all values in the warp ("uniform"), or differed from each other by a constant stride ("affine" e.g. 2,4,6,8), which was common for calculating addresses of loads. They suggested fine-grained clock-gating techniques to minimize power consumption when these patterns were detected.

Recently, several papers have been published exploiting the prevalence of zero values in graphics content. Zeroploit by Rangan et al.[333] takes advantage of the fact that operands of multiplications or logical ANDs instructions in shaders are frequently zero at run-time. They achieve a 35.8% improvement on average on selected individual shader pipelines, resulting in a 2.8% average improvement in frame-rate across several popular DirectX games and benchmarks on an NVIDIA GeForce RTX 2080 GPU. They use run-time value profiling based on Calder et al.'s original value-profiling work[318], but with a probabilistic least-frequently used approach to evicting values rather than periodically jettisoning half of the value profiling table. To simplify the transformations applied to the shader, value profiling instrumentation is injected at the IR level (called DXBC), instead of the low-level machine instructions used in other GPU value profilers[334][335].

Using the value profiling results, Zeroploit uses a cost-benefit heuristic to select shaders with significant portions of zero values as specialization candidates. Key "versioning" variables are selected to guard fast and slow paths depending on whether the value is zero or not at run-time. To generate the fast path, the zero value is propagated forward to the results of the multiplication or logical AND operation. It is also propagated backwards to eliminate the calculations of the other operand, which is no longer needed for this calculation with zero. Leobas et al. identified this type transformation as belonging to the class of "semiring optimizations" [336]. Shader bytecode specialized with this technique can then be injected back into the application at run-

time. Stephenson et al. extended this work to automatically select candidate shaders and versioning variables for Zeroploit by developing the PGZ tool[337] (formerly AZP [338]), which achieved similar performance benefits. Optimizations based on zero values are also applicable in other fields of GPU computation, such as for convolutions in deep neural networks[339].

3.5 Energy Efficiency in Mobile Games

The focus of this thesis is on improving the run-time speed of games, rather than optimizing for energy efficiency, but there are often overlaps between these objectives. Removing redundant calculations, for example, may speed up an application, and also reduce the energy it requires. Certain performance-vs-accuracy trade-offs also fall into this category of improving both speed and energy efficiency, at the cost of visual fidelity. Power consumption of video games, especially on battery-powered mobile devices, has a large and varied body of academic work. Many of these papers also include detailed workload characterisations, some of which use similar tooling to those used within this thesis. This section will explain the history of system-level approaches GPU power optimization, and various workload characterisations techniques that have evolved.

3.5.1 CPU DVFS for Software Rendering

In 2006, several researchers suggested that computer games would be a good candidate for DVFS (dynamic voltage and frequency scaling) . This is where the voltage and frequency of a processor are scaled up or down on the fly to reduce energy consumption during low-intensity periods, but provide sufficient resources to carry out calculations rapidly during high-intensity periods. DVFS had previously been successfully applied to video decoding [340] [341] [342] [343], as these workloads were computationally expensive, and had high variability. Researchers noted that video games had even higher variability, with some frames requiring orders of magnitude more work to render than others [344] [345].

Mochocki et al. suggest history-based performance prediction, basing the current frame's predicted performance on the previous recorded requirements within a certain window[346]. They also suggest an alternative approach using recorded signatures based on OpenGL ES state such as the number of triangles, and average triangle area

in each frame to predict power requirements of previous frames[347]. Gu et al. suggested that game-specific state such as the number and type of objects being rendered could estimate the rasterization workload of each frame, which acted as a good proxy for power and performance requirements [348]. In later work they demonstrated that DVFS using each frame's game state to estimate power consumption generally outperformed purely history-based techniques in terms of consistently meeting the target frame-rate, while still saving significant portions of system energy [349]. They also found success with control-theoretic techniques using a PID controller to scale processor voltage between frames [350], and in a hybrid approach combining both a PID controller, and a game-state-based prediction model[351]. Zhang et al. proposed a hierarchical approach using high-level game state, such as the current level, in-game story events, or scene changes to make higher-level power management decisions in addition to the PID and game-state based approaches at lower levels of the hierarchy. [352]. Wang et al. suggest a least-mean-squares linear predictor as a simpler and more generalized approach for estimating frame workloads, which sometimes outperforms PID controllers in gameplay scenarios where their gain parameters were not manually tuned to perform well in [353].

3.5.2 The Allure of Quake II

Much of the early research above focused on games using "software rendering", where the graphics calculations occurred on the CPU, as hardware acceleration via the GPU was not common for power-constrained mobile devices at the time. Many of the papers above used Quake II, a 1997 game by id Software, as a benchmark, as it was one of the few popular games whose source-code was readily available. This open-source nature made it popular among researchers, not just for power-management benchmarks, but for many other fields such as guiding GPU hardware architecture decisions [354], distributed systems architectures [355] [356], and artificial intelligence [357] [358] [359]. Quake II also contained the built-in capability to record and replay gameplay sequences, which made it appealing as a repeatable benchmarking tool, and precluded the need for external tracing application software of the type used throughout this thesis. However, as GPUs became increasingly common in consumer-level mobile devices, the CPU-based software rendering engine from Quake II became steadily less representative of modern gaming workloads over a decade after its release. As such, researchers had to find ways of implementing power-management schemes on games

that were no longer open-source, and made use of both the CPU and GPU.

3.5.3 Closed-Source Workloads on GPUs

Some researchers such as Vajus-Anttila et al. sought to overcome the lack of available game source-code by writing their own open-source rendering engine [360]. They used this to develop a mathematical model for predicting power consumption based on parameters such as the number of 3D model batches, number of triangles, and number of texels being addressed, and verified this via a fly-through scene of a virtual city. However, in order to gather these software metrics for power-estimation in real-time on real workloads, researchers had to find ways of gathering these sorts of metrics without source-code access.

Dietrich et al. sought to overcome the limitations caused by lack of game source-code access by instrumenting the Android operating system to intercept API calls indicating frame boundaries, and used time-stamps injected around each of these calls to implement a simple history-based DVFS scheme [361]. In later work, they used DLL-injection on Windows-based PC games to intercept graphics API calls (in this case DirectX 9) to enable workload characterization of more modern closed-source games [362]. This is a similar technique to the *apitrace*[12] tool used throughout this thesis. Their analysis showed that in Quake II, rendering accounted for $\sim 90\%$ of the CPU workload, whereas more modern games using hardware acceleration offloaded most of this computational complexity to the GPU, allowing more CPU time for more complex AI and physics simulations. They found that hardware-accelerated rendering experienced more inter-frame performance variability, and suggested an auto-regressive moving average technique for predicting frame workloads to avoid over-fitting problems and extensive manual parameter tuning found in prior PID-based techniques. Ma et al. found that on mobile game workloads, the GPU geometry processing stage of the pipeline was the main bottleneck, but around 30% of the frame time was spent on CPU-side game-logic such as path-finding[363].

Sun et al. showed that mobile games demonstrated high correlation between the number of textures being bound in each frame and the overall frame workload[364]. This is because reading and filtering large amounts of high-resolution textures was one of the more expensive operations, and more textures also correlated with more objects on screen, and more triangle area being rasterized. Using a low-overhead OpenGL ES wrapper to count the number of textures being bound each frame, they could use

this metric to govern DVFS models in a lower overhead and more accurate way than many prior workload prediction-based methods, and were able to save larger amounts of power in rapidly changing scenes. Dietrich et al. use textures in a different way, inferring gameplay state by comparing hashes of texture contents to infer whether textures being rendered in the current frame may indicate whether the game is in a high-interactivity gameplay scene, or a low-interactivity screen such as an options menu or level-select screen where the framerate and power consumption can be drastically reduced without impacting the user's experience [365] [366]. Cheng et al. use a similar DVFS scheme based on game-state detection, but instead use the number of memory allocations and textures being bound in each frame to infer whether the user is in a loading screen, low-interaction menu, or high-interactivity menu [367]. Mohammadi et al. [368] use apitrace [12] to intercept API calls, and estimate frame times based on the number of inputs to each pipeline stage.

3.5.4 DVFs for CPU, GPU, and Memory

Despite the shift towards using games with GPU-based rendering, the above work generally focuses on CPU-based DVFS. Pathania et al. introduce a scheme for governing both the CPU and GPU's voltage and frequency together in an integrated manner [369]. Park performed extensive workload characterization and developed various micro-benchmarks to motivate further work on DVFS taking into account the interactions between CPU, GPU, and shared memory system found on integrated mobile graphics chips [370]. Pathania et al. built upon their prior work by developing models to predict the performance interplay between CPU, GPU, and memory on a variety of mobile workloads, and use this model to guide a DVFS scheme to avoid power wastage [371]. Hsieh et al. also built co-operative CPU-GPU DVFS systems taking into account the impacts of their shared memory system [372].

The move to GPU-based rendering rather than CPU-based software rendering allowed for even more diverse power management opportunities [373]. You et al. suggest a software-based DVFS scheme separate from the GPU driver to allow operating systems or applications to dynamically adapt embedded GPUs' voltage and frequency in a manner independent of specific GPU manufacturers [374].

Tile-based architectures allowed for more granular power management options than the previous full-chip once per-frame schemes. Individual tile histories could be used to predict future workloads for sub-sections of each frame, allowing DVFS

schemes to throttle voltage and frequency on all but the most complex tiles of the frame being rendered [375].

The pipeline-based nature of graphics applications meant that shader cores could be selectively shut down during low-intensity pipeline stages to save energy if the workload bottlenecks could be effectively predicted [376]. In later work, they suggest additional power-gating strategies to shut down fixed-function geometry stages, and other non-shader units when pipeline stalls are detected [377].

3.5.5 Dynamically Varying Frame Rates

In addition to scaling the voltage of frequency of processors during different game states, another potential angle to achieve power savings is to scale the game's resolution or frame-rate dynamically based on factors such as the device's battery level, or the distance between the user and the screen [378]. Yan et al. suggest that frame-rates can be scaled down during "idle" periods of game-play when a certain period of time has elapsed without any user interaction events occurring [379].

Arnau et al. analyzed texture usage in mobile games, and found that 62% of all memory accesses were texture look-ups, and that 96% of texture data loaded in each frame was re-loaded again in the subsequent frame[380]. However, so much texture data is accessed each frame, that the look-ups cannot be efficiently cached between frames, as the re-use distance is so large. They propose rendering two frames in parallel to allow texture data to be shared between them. This trades application responsiveness for improved memory bandwidth utilization (and therefore energy efficiency), as the CPU-side game-state updates based on user inputs are inherently sequential, so the data for the two frames rendered in parallel is always slightly out of date.

In subsequent work, Arnau et al. also noted significant redundancy in fragment shader calculations, with around 40% of their calculation results being repeated on subsequent frames[381]. They suggest using hardware to augment their parallel frame-rendering technique with a hardware memoization scheme to record and re-use fragment shader results to improve energy efficiency. Keramidas et al. extend this scheme using a "clumsy value cache" which allows partial matches in the memoization table by reducing the precision of the input parameters [382]. Most relevant to this thesis is their analysis of how varying floating point precision of various fragment shader instructions impacts the image's visual quality, where they find that values used as indices into textures are far more sensitive to precision reduction than many other cal-

culations.

3.5.6 Avoiding Overdraw

In addition to their work on variable frame rates, Yan et al. also identified significant power wastage due to large amounts of overdraw occurring in mobile games[379]. Overdraw is when the same pixel is rendered multiple different times, with later results invalidating the prior pixel colour results computed earlier in the frame. The performance impact of unnecessary overdraw was well-known[383], and minimizing the overheads of overdraw was one of the benefits of the now-prevalent tile-based architectures used in mobile GPUs [384]. Prior work had estimated overdraw for mobile 3D games to be around 2-3X [385], but Yan et al. showed that many popular mobile games, especially 2D games vastly exceeded this estimate to an average of 15.5x (and up to 157.9x in some cases). Many of these games were ignoring well-established game optimization techniques such as sorting and rendering objects front-to-back, or performing a depth pre-pass[386], both of which would allow early depth-buffer testing to reject many pixels early using the well-known Z-buffer algorithm[116] [117] baked into all modern hardware. A potential hardware-based solution to this problem was presented by de Lucas et al., in which they attempt to enforce visibility rendering order (front-to-back) by using results from depth-tests in previous frames to predict the relative order to render objects in the current frame [387].

3.5.7 Variable Floating-point Precision

Another development prevalent around mobile GPUs is the concept of variable floating-point precision. Hao et al. showed that many common lighting and vertex transformation calculations can be performed as reduced fixed-point precision calculations, with very little visual difference from the full floating-point calculations [388]. These reduced-precision vertex calculations can efficiently exploit SIMD data-paths [389]. Reduced precision texture look-ups can also provide power-savings [390]. Akely et al. explored the bounds to which 3D position data could have its precision reduced while avoiding artefacts known as "z-fighting" (where insufficiently precise depth information is stored for non-overlapping objects, such that they appear overlapping and flickering through one another) [391]. Pool et al. applied these ideas to reduced-precision calculations in arbitrary vertex shaders[392] and fragment shaders [393], leading to reduced power costs of both calculation and data transmissions [394]. Systems for

dynamically selecting precision using fine-grained power-gating were also examined [395]. Several systems have also been explored to dynamically balance between image quality and energy consumption at run-time by switching between reduced floating-point and fixed-point precision levels either per scene [396], per frame [397], or per program [398].

3.6 GPU Debugging and Profiling

Debugging and profiling GPU programs is a complex task which requires specialized tools. Some sources of this complexity are the inherently parallel nature of GPU computations, limitations on older hardware around control-flow and interrupts, and the interaction between closed-source graphics drivers and undocumented hardware architectures. This section explores many of the current industry standard tools for this, as well as some of the academic research conducted in this field.

3.6.1 Current Industry Tools

Because each GPU vendor uses their hardware performance counters, in-house ISA, and driver stack, they often provide custom debugging and profiling tools. Such tools include NVIDIA's NSight[399], AMD's Radeon Developer Tools Suite[400], Intel's Graphics Performance Analyzers[401][402], ARM's Graphics Analyzer[403], and Qualcomm's Snapdragon Debugger[404] and Profiler[405].

Operating system vendors also release tools for debugging and profiling graphics applications, such as Apple's Metal system trace[406] or frame capture debugging[407] tools, or Microsoft's PIX debugger for DirectX 12[408]. Microsoft also released an OS-level event profiler for GPU programs called GPUView[409][410] to help measure and visualize events between the CPU and GPU. Valve's open-source GPUVis[411] tool provides similar visualization of CPU-GPU interactions within graphics applications on Linux.

In addition to these OS-level or vendor-specific tools, there are also several popular open-source tools aiming to aid in debugging graphics applications that focus on the API-level. Baldur Karlsson's Renderdoc[412] allows capturing, visualization, and exploration of all graphics state in a single frame, and allows for step-by-step playback of individual calls in that frame to view how different buffer contents are affected. It is compatible with DirectX 11 and 12, OpenGL, OpenGL ES, and Vulkan across a

variety of operation systems. Renderdoc's primary focus is to aid in debugging incorrect rendering results, but it also contains some limited profiling capabilities to detect performance hot-spots within captured frames. Other debugging tools, such as José Fonseca's apitrace[12], aim to capture and replay whole traces of rendering API calls. Apitrace allows tracing of OpenGL and older DirectX dialects, whereas newer tools by LunarG such as vktrace[413] and its successor GFXReconstruct[414] aim to provide similar tracing support for Vulkan applications.

3.6.2 GPU Debugger Research

In addition to the rapidly-evolving array of industry and open-source developed tools, graphics profiling and debugging tools have been the focus of numerous explorations within academia. The utility of using graphical visualizations to debug the behavior of massively parallel programs was explored in the 1990's via tools such as IVE[415] and Prism[416] before the dawn of GPUs. This idea is even more powerful in graphics applications, where many of the calculations are inherently visual in nature.

In 2005, Duca et al. proposed a relational debugging engine[417], which used SQL-like queries on graphics state, and could visualize the query results in a variety of ways. This debugging engine was based on intercepting OpenGL calls within a cluster-based distributed rendering framework called Chromium[418], and storing them in a database. In 2007, Strengert et al. proposed a hardware-aware debugger[419] that would allow data to be retrieved from real GPU hardware using a combination of OpenGL API-call interception, shader instrumentation, and OpenGL extensions for accessing shader outputs on the CPU. Sharif et al.'s "Total recall" debugger[420] used partial emulation on the CPU to allow developers to easily step through and add break-points to shaders, as well as adding a technique for tracking and examining the history of pixel values in intermediate buffers used in multi-pass rendering techniques. Van Dyk et al.[421] extended the scope of debugging techniques from capturing and visualizing the current state, to capturing the state's entire history, and using a custom query language to compare state at different points in time.

Hou et al.[422] proposed a method for debugging GPGPU programs by instrumenting GPU-side kernels with code for suspending themselves and recording their state, falling back to a CPU-side interrupt handler, and then resuming the kernel's execution. This approach allowed the CPU to inspect, record, and visualize the kernel's intermediate state and then appear to seamlessly restart its execution from the current point,

even though hardware-based interrupts were not available on GPUs at the time to let them natively call CPU-side interrupt-handling code.

3.6.3 GPU Profiling & Performance Estimation Research

There has also been significant work on profiling and performance estimation techniques. Wimmer et al. explored the use of heuristic functions to estimate the rendering time of different 3D scenes[423]. In 2005, Moya et al. developed a GPU simulator called ATTILA[85], and used its performance results to demonstrate the benefits of a unified shader architecture, compared to the separate vertex and fragment units present in GPUs at the time. Bakhoda et al.[424] also used a similar simulator-based approach to analyse various CUDA workloads with GPGPU-Sim.

Stephenson et al. proposed a compile-time instrumentation tool called SASSI[334] to inject handlers for various workload characterization and profiling tasks into CUDA kernels, allowing for larger datasets to be explored without the performance limitations of running on a simulator. CudaAdvisor by Shen et al.[425] takes a similar approach, but uses the open-source LLVM framework rather than NVIDIA's in-house compiler, and instruments both the CPU-side and GPU-side code. Nvbit by Villa et al. [426] also offers similar capabilities to SASSI, but uses dynamic binary instrumentation, which removes the need for source-code access present in compile-time approaches, and allows kernels JIT-compiled by the driver to be instrumented.

3.7 Summary

The preceding sections have covered a wide range of research, and provided background on GPU shader programs, tools to analyze them, and compilation techniques which may be used to simplify and optimize them for both power and performance.

Section 3.2 explained the evolution of programmable shaders from their initial purpose of providing flexibility to CPU-based raytracing systems, to the current GPU-accelerated programmable vertex and fragment shader pipeline. It also covered the numerous additional types of shaders, and general-purpose computing programs now possible on modern programmable GPUs due to their unified shader architecture.

Several approaches to simplifying shader source code were explored in Section 3.3. These usually took place as part of level-of-detail (LoD) systems, which exploit the fact that calculation inaccuracies for far away objects are usually imperceptible in the

realm of graphics.

[Section 3.4](#) covered techniques for optimizing code based on knowing which values were in use. These included constant-folding of statically known values, hardware-based branch or value predictors, code specialization based on value profiling, and run-time techniques such as JIT compilation and dynamic binary translation.

[Section 3.5](#) explained the history of profiling and workload characterization for graphical applications for the purpose of increasing their energy efficiency. The transition from open-source CPU-based workloads to closed-source GPU-accelerated workloads was discussed, along with how workload profiling tools had to evolve as a result. Several optimization strategies were also explored, such as reducing overdraw, or lowering the floating-point precision of calculations to improve both performance and power-efficiency for trade-offs in visual quality.

Finally, the current state of GPU debugging and profiling tools was covered in [Section 3.6](#), along with some of the academic work that led to these developments. This included current GPU-vendor-specific toolsets, along with cross-platform open-source tools such as apitrace[12] which is used throughout this thesis.

Chapter 4

Compiler Optimizations for Individual Shaders

4.1 Introduction

This chapter investigates the impact of various traditional compiler optimization techniques on shaders[9]. Using a source-to-source GLSL shader compiler framework called LunarGlass[10], the impacts of various combinations of different optimizations were examined on several different mobile and desktop GPUs from different hardware vendors. For some fragment shaders, performance improvements of up to 25% were achieved, but these results were not universal across all vendors, and some combinations also resulted in slow-downs. Using an iterative compilation approach, the best combinations of optimizations were determined for each shader on each GPU, demonstrating that the correct sets of optimizations are not universal, and should be determined carefully on a vendor-to-vendor basis.

[Section 4.2](#) provides a motivating example where a combination of source-level optimizations provide significant speed-ups. In [Section 4.3](#), the compiler tools and optimization passes used are discussed in greater detail. The experimental fragment shader timing tools developed are discussed in [Section 4.5](#), along with a characterization of the shaders used as benchmarks. Finally, the results of iteratively compiling and timing each fragment shader are reported in [Section 4.6](#).

4.2 Motivating Example

```

in vec2 uv;                out vec4 fragColor;
uniform sampler2D tex;    uniform vec4 ambient;
/*Main Function*/
//9 symmetric weights and texture sampling offsets
const vec4[] weights = vec4[](vec4(0.01), ... , vec4(0.01));
const vec2[] offsets = vec2[](vec2(-0.0083), ... , vec2(0.0083));
float weightTotal = 0.0;
fragColor = vec4(0.0);
for(int i = 0; i < 9; i++){
    weightTotal += weights[i][0];
    fragColor += 3.0*ambient*weights[i]*texture(tex, uv+offsets[i]);
}
fragColor /= weightTotal;

```

Listing 4.3: Example shader before optimization

[Listing 4.3](#) illustrates an example fragment shader where source-to-source compiler optimizations such as loop unrolling and arithmetic reassociation give performance improvements of up to 45% (see [Figure 4.1](#)). This simplified example is based on a GFXBench 4.0[8] fragment shader which repeatedly samples from a 2D image texture `tex` at various constant offsets from the coordinates `uv` (passed in from the vertex shader). The weighted sum of these samples is written to `fragColor` as the final RGBA pixel colour output. Repeated weighted texture sampling code like this is common for various blur or glow effects in games.

```

/*Main Function*/
vec4 fc1 = texture(tex, uv+vec2(-0.0083));
...
vec4 fc9 = texture(tex, uv+vec2(0.0083));
vec4 t0 = fc5 * vec4(1.83);
vec4 t1 = (fc4 + fc6) * vec4(0.63);
vec4 t2 = (fc3 + fc7) * vec4(0.42);
vec4 t3 = (fc2 + fc8) * vec4(0.15);
vec4 t4 = (fc1 + fc9) * vec4(0.03);
vec4 sum = t4 + (t3 + (t2 + (t0 + t1)));
vec4 fac = vec4(0.699301) * ambient;
fragColor = sum * fac;

```

Listing 4.4: Example shader after optimization

This code has many optimization opportunities, as can be seen in [Listing 4.4](#).

Firstly, it contains an unrollable loop with 9 constant iterations. Once unrolled, the sum for `weightTotal` contains only constants, so can be completely evaluated at compile time. The `weightTotal` can also be inverted before multiplying it by `fragColor`, thereby changing 8 additions plus a division into a single multiplication.

Each value summed for `fragColor` has a common multiple of $3.0 * \text{ambient}$ which can be factorised out, leaving 1 multiplication instead of 9. The constant 3.0 can then be folded into the `weightTotal` before multiplying it by `fragColor` too, resulting in a single constant factor of 0.699301 to multiply.

Because the weights are symmetric, pairs of texture samples will share weights as common multiples which can be factorised out too. These factorizations are technically unsafe to perform, as floating-point arithmetic is not associative. However, for most use-cases within shaders like this, the visual impacts will not be noticeable, and the slight reduction in correctness is outweighed by the improved performance.

Another benefit of unrolling the texture-sampling loop is that all the texture sampling calls now use a simple constant offset such as `uv+vec2(0.0083)`, rather than an offset that is dependent on a loop index such as `uv+offsets[i]`. This constant offset can provide benefits such as improved texture pre-fetching or caching, and modifying this in the source code means these benefits are no longer contingent on the GPU vendors' compilers unrolling the loop or detecting these constant offsets internally.

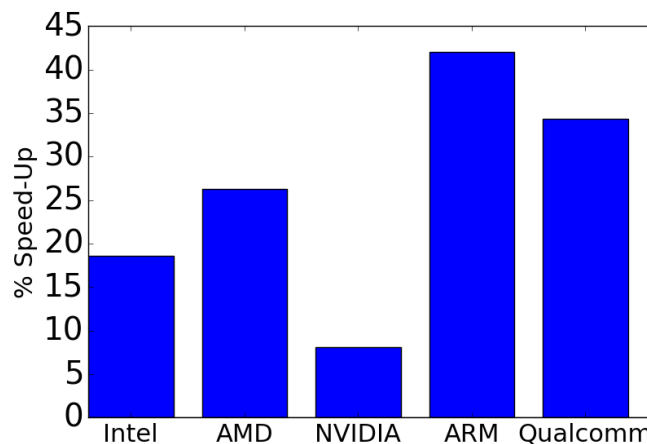


Figure 4.1: Performance gains from optimizing example shader on each platform

For this example, optimizations provide large performance impacts, with speed-ups of 7-28% on desktop, and 35-45% on mobile (see [Figure 4.1](#)). This means GPU vendors' compilers do not catch every optimization opportunity, and offline compilers can have a large impact. It also shows that performance impacts can vary drastically depending on which GPU is used. Some vendors may not perform extensive loop

unrolling to reduce shader compilation times which may lead to noticeable delays. The use of extensive unsafe floating point arithmetic reassociations is also something that driver-based optimizations may be unable to do as they must preserve a greater level of correctness, but can be leveraged by developers via offline source-to-source compilation if they determine they do not require such strict correctness constraints.

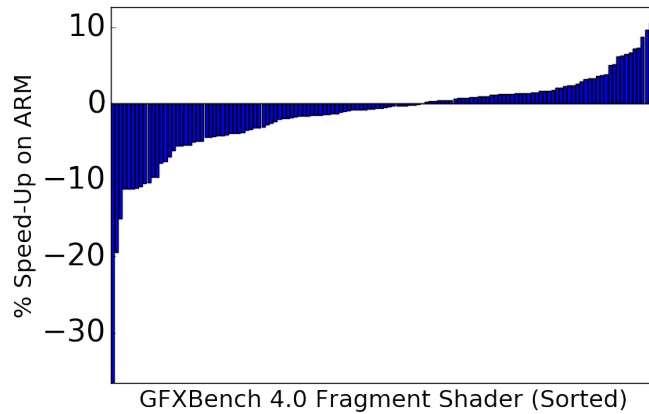


Figure 4.2: Distribution of performance impacts on an ARM GPU when all optimizations are enabled (positive = speed-up, negative = slow-down). Each bar in the x-axis represents a different individual shader extracted from GFXBench 4.0 (sorted by impact).

Despite this universal positive impact of optimization on the example shader in Listing 4.3 on all platforms seen in Figure 4.1, applying these optimizations to all fragment shaders in the GFX 4.0 benchmark, gives very variable performance results. In Figure 4.2, ARM’s Mali-T880 gains up to 10% and loses up to 30%. This shows us improvement is possible in real-world shaders, but a one-size-fits-all approach can do more harm than good. Smarter techniques to choose when and how to optimize each shader for each platform are necessary to reap the performance rewards but avoid the large performance pitfalls.

4.3 Example Optimizations

This section describes the source-to-source optimization techniques explored, and the tools used to do so.

4.3.1 LunarGlass Optimization Framework

To perform the source-to-source optimizations on GLSL[36] shader code, the LunarGlass framework from LunarG was used. This is a modified version of LLVM

3.4[427], with several extensions for GLSL-specific intrinsic functions, optimization passes, and a GLSL front and back end. The default optimization passes which can be toggled via command-line flags are described below.

- **ADCE** - Aggressive dead code elimination.
- **Hoist** - Flatten conditionals by changing assignments inside "if" blocks into select "select" instructions.
- **Unroll** - Simple loop unrolling for constant loop indices.
- **Coalesce** - Change multiple individual vector element insertions into a single swizzled vector assignment.
- **GVN** - Global value numbering.
- **Reassociate** - Reorder integer arithmetic to simplify it (or some floating-point expressions such as $f \times 0$).

Several other LLVM optimizations were included, such as constant folding, common sub-expression elimination, and redundant load-store elimination. However, these passes were not altered within these experiments, as they were not exposed by default via command-line flags, and some were necessary to canonicalize instructions for future optimizations.

After combining these pre-existing passes with some additional passes ones to handle unsafe floating point arithmetic (see below), iterative compilation was used to explore their impacts. Because only 8 flags were available, it was possible to exhaustively apply all 256 possible combinations of passes. Many of these resulted in duplicated source code (see [Section 4.4](#)), so measuring the performance impact of all these generated outputs was tractable in this case. It may be possible to use results from this sort of exhaustive analysis to guide better flag selection heuristics or machine learning in future work.

4.3.2 Additional Unsafe Optimizations

In addition to LunarGlass's default passes, several custom unsafe floating point optimization passes were also implemented. Many of these mimicked parts of the integer reassociation pass to perform simple arithmetic simplifications such as:

$$ab + ac \longrightarrow a(b + c)$$

$$a + a + a \longrightarrow 3a$$

$$a + b - a \longrightarrow b$$

Arithmetic operations were also re-ordered to group constants together for better constant folding and propagation, and to group scalar operations before turning the results into vectors. This scalar reassociation was designed to minimize unnecessary registers slots holding temporary vector results, when single scalar registers would suffice:

$$f_1(f_2v) \longrightarrow (f_1f_2)v$$

$$c_1(c_2v) \longrightarrow (c_1c_2)v$$

Where f_1, f_2 are scalar floats, and c_1, c_2 are constants. This re-ordering also canonicalized the sequence in which the operands occurred, which could allow for greater common-sub-expression elimination opportunities in subsequent passes.

Other identities such as multiplying by 1, or adding 0 were also optimized out, and division by constant operands was changed into multiplication by the operand's inverse (which could be determined at compile time).

$$f_1 \div c_1 \longrightarrow \frac{1}{c_1} \times f_1$$

$$0.0 + f_1 \longrightarrow f_1$$

$$1.0 \times f_1 \longrightarrow f_1$$

Sums containing negated pairs of values were also eliminated:

$$a + b + (-a) \longrightarrow b$$

To ensure the maximum number of these patterns could be detected, a pre-processing step was run to transform all subtractions into additions of a negated value, and all divisions into a multiplication by an inverse:

$$a - b \longrightarrow a + (-b)$$

$$a \div b \longrightarrow a \times \frac{1}{b}$$

The aim of these additional passes was to explore the impact of unsafe floating point optimizations which could not be implemented in a conformant GPU driver's compiler, but would fit well in an offline optimization framework where the developer can control when they are used.

4.3.3 Artefacts

Because OpenGL drivers only accepted shaders as GLSL source-code (until the recent SPIR-V extension was standardized in OpenGL 4.6), source-to-source transformations were the only option for a cross-platform investigation of shader optimization. However, this led to artefacts that would not occur in typical human-written GLSL code, and could sometimes negatively impact the code's performance. Such artefacts included:

Scalarized Matrix Multiplications

GLSL has primitive types for both vectors and matrices, and humans may write code such as:

```
mat4 m1, m2; vec4 v;
mat4 m = m1 * m2;
vec4 res = m * v;
```

When this is processed in LunarGlass, however, the matrices are divided up into their individual scalar components, and instead of 2 lines of matrix-vector calculations, tens of lines worth of scalarized calculations will be generated in LunarGlass's output GLSL.

Multiplication of 4×4 matrices is very common in GLSL, especially for vertex shaders, so GPU driver compilers are used to handling this use-case. However, completely scalarizing these calculations in the source code can obfuscate the fact that a matrix multiplication is taking place. This can affect the order and batching of load-/store operations for retrieving the matrix's elements, and reduce performance by obscuring the code's purpose from the GPU vendor's compiler.

Unnecessary Vectorization

In LLVM, operands for addition, multiplication etc. must be of the same type. This means when adding or multiplying a vector, the operands must both be vectors. In GLSL, the syntax allows you to multiply both vectors and matrices by scalars:

```
vec4 v;
vec4 res = v * 0.3;
```

Since LunarGlass is based on LLVM, it has to vectorize these floating point values before multiplying them, resulting in unnatural output code like:

```
vec4 v;
vec4 C_vec4p0p3 = vec4(0.3, 0.3, 0.3, 0.3);
vec4 res = v * C_vec4p0p3;
```

This unnecessarily increases the number of vector constants and vectorization instructions, so may affect the amount of registers or constant storage memory for shaders if the driver's compiler is unable to detect this case.

Large Basic Blocks

The conditional flattening and loop unrolling passes result in very large basic blocks in the generated code. This can put pressure on the register allocators in the GPU vendor's compiler.

These control flow reduction passes coupled with the massive number of instructions the scalarized matrix multiplication generates can lead to huge basic blocks with hundreds of instructions, which may be difficult for compilers to handle well.

Mobile Shaders

In order to run the desktop OpenGL shaders on mobile devices (which use OpenGL ES[21]), they were first converted into SPIR-V using glslang[428], and SPIR-V Cross[429] was then used to generate GLES compatible shaders.

$$GLSL_{OpenGL} \longrightarrow \mathbf{LunarGlass} \longrightarrow GLSL_{OpenGL} \longrightarrow \\ \mathbf{glslang} \longrightarrow SPIR-V \longrightarrow \mathbf{SPIRV-Cross} \longrightarrow GLSL_{OpenGLES}$$

Having passed through so many compilation tools means the code picked up slight quirks and artefacts from each one in turn, and was often very different from the orig-

inal desktop GLSL shader. As a result, some of the measurements on mobile may be impacted by artefacts that are not present on desktop.

4.4 Benchmark Characteristics

For the timing experiments, fragment shaders were chosen from GFXBench 4.0, a graphics hardware benchmarking suite from Kishonti [8]. It was designed as a standard way to compare the real-time rendering performance of GPUs from different vendors. The OpenGL version of GFXBench 4.0 contains several 3D animated scenes designed to use advanced and expensive rendering techniques to test the GPU's capability under heavy loads. There are also several lower-level tests to stress specific aspects of the hardware individually.

Performance on these benchmarks is important to vendors because they are used to compare against GPUs from competitors. GFXBench 4.0 was chosen here as a benchmark suite because it is a well-known, self-contained, cross-platform benchmark that covers a variety of different situations of varying complexity to test out shader compilation techniques.

This section will describe the various benchmarks within the GFXBench 4.0 suite, explain how shaders were extracted, and characterise some aspects of the different shaders' complexities.

4.4.1 Benchmarks within GFXBench 4.0

The GFXBench 4.0[8] benchmark suite contains several sub-categories of graphics benchmarks, including battery life and rendering quality tests. However, the two categories of benchmarks used as performance benchmarks within this chapter are the high-level tests, which render complex animated scenes, and the low-level tests which render simpler scenes designed to stress individual aspects of rendering performance.

4.4.1.1 High-Level Tests

The high-level rendering tests all run real-time animated scenes for around a minute, and aim to use numerous computationally intensive rendering techniques to push the GPU hardware to its limit. Each test features both an on-screen and off-screen variant. The on-screen version renders the final image full-screen at the device's native monitor resolution, whereas the off-screen version uses a fixed 1080p image resolution. This



(a) Manhattan



(b) Manhattan 3.1



(c) Car Chase



(d) T-Rex

Figure 4.3: GFXBench 4.0 high-level benchmarks

ensures that rendering performance results can be meaningfully compared between GPUs no matter what the resolution of the attached screen is. However, both versions use identical shaders, so only one version of each test is analysed in this chapter.

Manhattan Originally introduced in GFXBench 3.0, the Manhattan benchmark features several helicopters patrolling a ruined sci-fi city-scape at night-time, before eventually confronting the giant chrome-plated robot that was the cause of this destruction. The night-time city scene showcases numerous dynamic lighting effects from glowing billboards, fire, and searchlights, as well as full-screen post-processing effects such as bloom[430] and depth-of-field[431].

Manhattan 3.1 Enhancing the original Manhattan benchmark, this version replays roughly the same scene, but with enhanced visual effects, such as dynamically simulated lightning and electrical effects using compute shaders and indirect draw calls. This version also uses high dynamic range (HDR) rendering[432] to give an enhanced contrast between light and dark areas of the scene, and improve effects such as bloom[433].

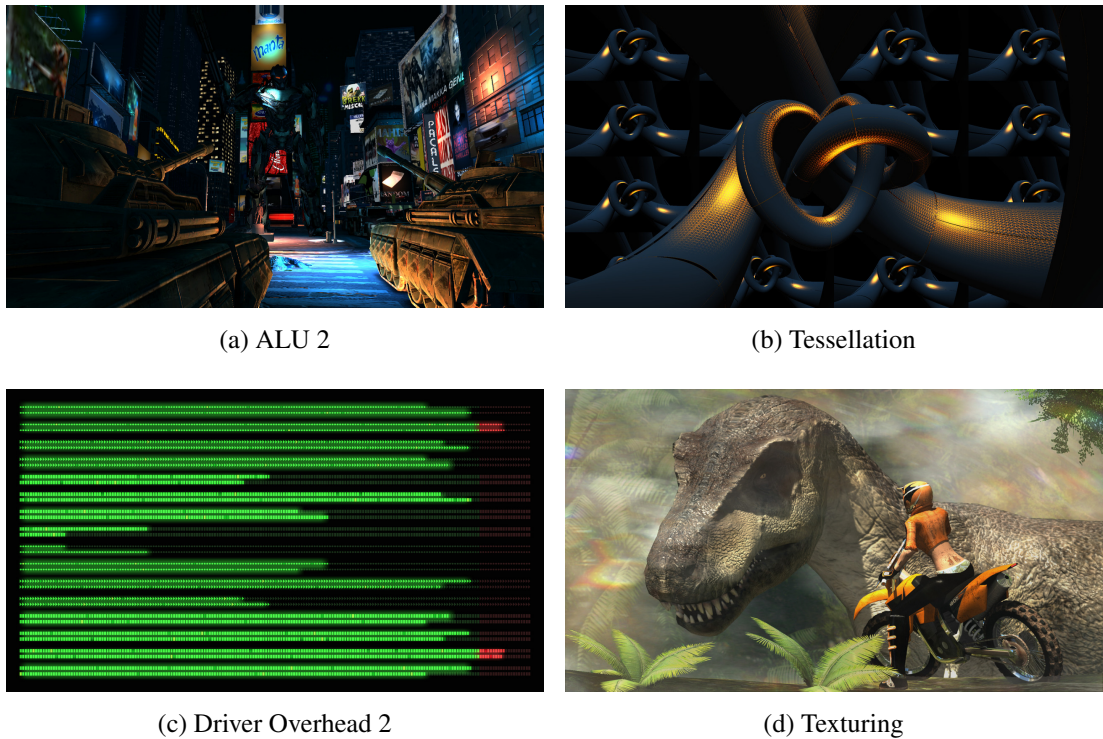


Figure 4.4: GFXBench 4.0 low-level benchmarks

Car Chase New in GXBench 4.0, this benchmark features cinematic shots of a car chase between two sports cars on a coastal road around a tropical island. It makes use of newer shader types such as geometry, tessellation, and compute shaders for numerous complex effects. They also make use of physically-based rendering[434][13], a lighting technique now common in many modern games, as well as numerous post-processing effects such as screen-space ambient occlusion (SSAO)[19], lens flare[435][436], and motion blur[16][17].

T-Rex Originally introduced in GFXBench 2.7, this scene features a motor-biker being chased by a t-rex through dense jungle vegetation. This benchmark targets an older version of OpenGL than the rest, so does not make use of as many complex lighting and post-processing techniques, but still features a post-processing motion blur[16][17] pass, as well as detailed textures and model geometry.

4.4.1.2 Low-Level Tests

GFXBench 4.0's low-level tests are designed to stress individual aspects of the rendering pipeline. As with the high-level tests, both on-screen and off-screen variants of all

four tests are available, but the shaders used within both are identical.

ALU Designed to focus on the fragment shader workload from the Manhattan benchmark, this benchmark repeatedly calculates the dynamic lighting for a single scene with static geometry.

Tessellation Using simple vertex and fragment shaders, this benchmark is designed to measure the performance of tessellation shaders. It renders some animated knotted tori using Bézier surfaces, which are tessellated in real-time using tessellation shaders[437].

Driver Overhead This test is designed to measure the CPU-side overhead of the vendor's OpenGL driver implementation when running the Manhattan benchmark. It renders double the amount of geometry from Manhattan, with additional draw calls, context switches, and state changes to add additional CPU-side load for the driver to work around.

Texturing This approximates the texturing load of the Manhattan benchmark by using numerous layers of overlapping textured quads. These textures are designed to have the same format and compression status as those in Manhattan to isolate and measure the GPU's texture look-up performance.

4.4.2 Extracting Shaders

GFXBench 4.0 is a closed-source application. However, OpenGL requires shaders to be submitted to the API as GLSL source-code strings, rather than as compiled binary or intermediary representations like other APIs (see [Section 2.3](#)). This means that if accessing the graphics API calls would allow these shader source strings to be intercepted and used to test source-to-source optimization tools on.

As the majority of real-world graphics applications are closed-source software, instrumenting and extracting information from such applications is an area that many graphics systems researchers have had to contend with (see [Subsection 3.5.3](#)). A common solution to this problem is to perform some sort of DLL-injection technique to intercept the OpenGL API library calls, record them for analysis, and then pass them on to the real API OpenGL API library. A technique similar to this approach is used

in subsequent chapters, but for the shader-only analysis performed in this chapter, a simpler approach suffices.

As GFXBench 4.0 is compatible with Linux, it can run on a device using an open-source Mesa graphics driver[438]. Mesa graphics drivers have a built-in functionality for dumping shader source code from any application running on them by setting environment variable:

```
MESA_SHADER_DUMP_PATH=./dump/shaders
```

By setting the above environment variable, and simply running the full GFXBench 4.0 suite, it is possible to extract the source-code for all shaders within it.

4.4.3 Deduplicating Shaders

Some shaders are re-used between the benchmarks, especially the different Manhattan versions and the low-level benchmarks that are based on subsets of it, so deduplicating the shaders is useful to avoid over-representing them in subsequent analysis. When Mesa dumps shaders, it writes them to files named using a hash of the source-string's contents, so an initial deduplication pass can be trivially performed by simply rejecting shaders with duplicate file names. However, this is not always sufficient.

Many of the benchmark's shaders follow the “übershader” pattern[439], where a single file containing numerous graphics techniques is customised via preprocessor directives to enable or disable sections when generating shader instances. As such, some shaders are identical apart from preprocessor `#define` statements, forming families of similar shaders where some optimizations apply frequently because all include the same code segment, despite being specialized in different ways elsewhere.

Some shaders that differ only in pre-processor directives are functionally identical, as some of the preprocessor directives are irrelevant to the actual code. To avoid this from skewing the benchmark shaders, a more complex process was adopted, consisting of several normalizing transformations before finally deduplicating the shaders.

$$GLSL_{GL} \longrightarrow \text{preprocess} \longrightarrow \text{glslang} \longrightarrow SPIR-V \longrightarrow \text{SPIRV-Cross} \longrightarrow GLSL_{ES}$$

The first part of the transformation pipeline involved several preprocessing steps. These included adding missing version numbers, removing line-continuations, or renaming outdated keywords (e.g. the `varying` keyword becomes `in` or `out` from GLSL 1.30 onwards). Khronos's glslang[428] compiler was then used to transform the shader

into SPIR-V[37], which was then submitted to SPIRV-Cross[429] to convert it into GLSL for OpenGL ES.

This had several benefits, such as removing all preprocessor directives, as well as normalizing the formatting and some of the variable names. This meant that many of the superficially different but functionally identical shaders now had identical source-code. After a second hash-based deduplication pass, the number of remaining shaders was drastically reduced (as shown in Table 4.1), which ensured that all shaders being experimented now genuinely contained different code.

Of the 165 remaining non-duplicate fragment shaders, 10 of these merely return a constant value, so are uninteresting for compiler optimization experiments. A further 8 shaders do not work with the LunarGlass compilation tool, so are also excluded from tests. As such, the final benchmarks used throughout this chapter are the 147 fragment shaders extracted from GFXBench 4.0 that are non-duplicates, non-constant, and function correctly within LunarGlass.

Throughout the rest of this chapter, all experiments are performed on the shaders remaining after this deduplication step.

Deduplication Method	Fragment	Vertex	Tessellation Control	Tessellation Evaluation	Geometry	Compute
None	698	675	13	14	2	40
File hash	570	558	13	14	2	40
GLSL ES	165	120	9	9	2	40

Table 4.1: The number of shaders of each type from GFXBench 4.0 after different deduplication schemes

4.4.4 Shader Characteristics

Graphics shaders are somewhat different in nature from typical CPU code or other forms of GPGPU code. Here, the nature of the benchmark shaders is examined, including their typical complexity, and their susceptibility to the optimization passes.

4.4.4.1 Static Code Size

The number of lines of code shown in Figure 4.5 can be used as a rough proxy for the shaders' complexities. As GFXBench makes extensive use of `#define` and `#ifdef`

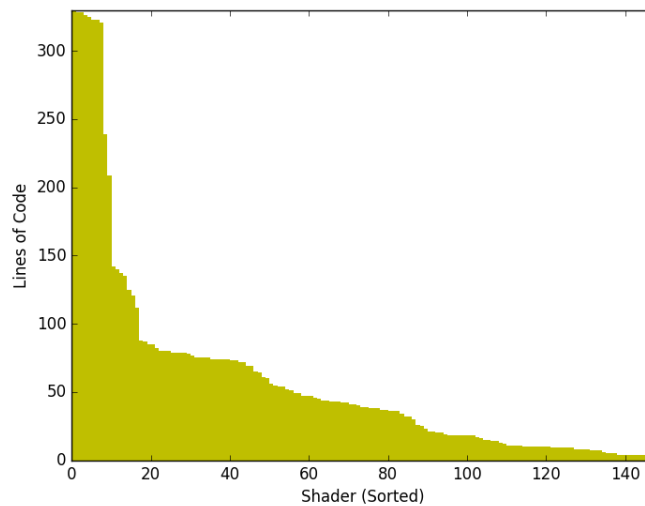


Figure 4.5: Lines of code for GFXBench 4.0 fragment shaders (after preprocessing). Comments, white-space, lone brackets, and uniform/input declarations are ignored in these counts.

directives to compose its shaders, the code was passed through a C preprocessor to remove these macros. Due to the “übershader”[\[439\]](#) pattern used by many of the shaders, this preprocessor pass eliminates many unused lines and functions that were included in the source code but guarded by macros, allowing the number of executable lines in the real instantiated shader to be measured more accurately. However, not all unused functions are eliminated by the preprocessor, as it does not perform a semantic-based dead-code elimination pass, so the “lines of code” metric is only suitable as a rough proxy for complexity.

[Figure 4.5](#) shows that the shaders’ line-counts form a rough power-law distribution, with very few lengthy shaders, and numerous simpler shaders. Unlike typical C-code, the most complex shaders do not exceed 350 lines, with the most containing fewer than 50, and some containing a mere handful of simple instructions.

When examining the shaders’ source-code, it is clear that this simplicity extends beyond line-counts. Complex control-flow is uncommon within these shaders, with few containing any loops. Even branching instructions are limited, with most shaders containing no more than 2 `if` statements if any. This simple control flow often leads to larger basic blocks than would be found in typical CPU-side code. These blocks often contain long sequences of arithmetic instructions punctuated with texture look-ups. This is partly due to the types of algorithms typically expressed within shaders, and partly due to performance concerns, as branching instructions were historically very

costly on older GPUs, especially if they caused divergent behavior between parallel shader invocations.

4.4.4.2 Dynamic Cycle Count

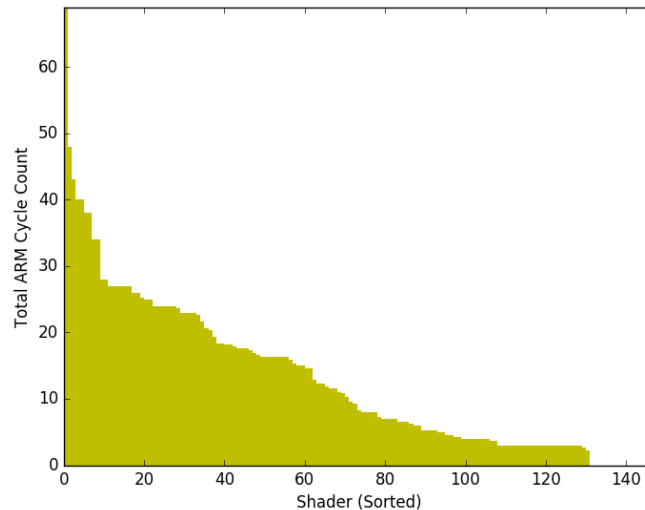


Figure 4.6: Sum of all cycles spent on Arithmetic, Load/Store, and Texture operations on the longest execution path (From ARM’s offline shader compiler).

An alternative method of estimating complexity is to use a cycle-count prediction tool such as ARM’s offline shader compiler[440] for Mali GPUs. This metric has the benefit of eliding unused function definitions, and measuring only instructions that are executed. However, the estimated counts here are platform-specific, so may not generalize to other GPUs, but can still provide a rough proxy for relative shader complexities.

The toolchain described in [Subsection 4.4.3](#), was used to convert the desktop OpenGL shaders into mobile OpenGL ES shaders compatible with ARM’s static analysis tools. Due to certain control flow features such as dynamic cycles, some of the 147 benchmark shaders have no predicted cycle-counts, but the majority were statically analyzable. Shaders featuring these more dynamic control flow patterns typically had high line-counts too, so would likely have higher cycle-counts than many of the other shaders if measured dynamically.

The estimated ARM cycle counts for the longest execution path in each shader shown [Figure 4.6](#) feature a less pronounced power-law-like shape than the line-count metric in [Figure 4.5](#). Both graphs have long tails with a large number of low-complexity shaders. These simple shaders offer fewer opportunities for compiler optimizations, as

there are only a limited number of lines of code to deal with, and therefore a lower probability of finding instructions that can be optimized. However, there are still a sufficient number of many-cycle shaders (including those missing from the graph due to not being statically analyzable), that further optimizations may prove useful.

4.4.4.3 Uniqueness

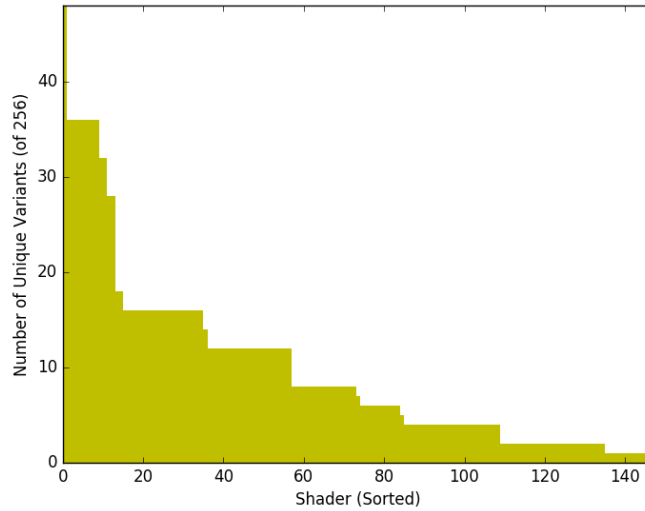


Figure 4.7: Number of unique shader variants generated from all possible combinations of LunarGlass and custom passes.

Another complexity metric relevant to the compilation tools used here is the number of unique variants generated from different combinations of compiler passes. In LunarGlass, the selected optimizations are toggled on or off via 8 boolean flags, resulting in $2^8 = 256$ possible combinations. However, not all optimizations are applicable to all shaders, or may produce identical results to other combinations. For instance, a shader with no loops will generate the same optimized code no matter whether loop unrolling is enabled or disabled.

By iterating through all 256 combinations of optimizations, and then running a file-hash based deduplication program, the number of unique shader variations can be quantified. The number of unique variants generated for each shader is shown in [Figure 4.7](#).

Compared to the 256 possible optimization variants, the number of unique shaders generated is significantly lower. Few shaders exceed 10 distinct variants, with the highest reaching only 48. As explained above, most shaders are very short and simple, with low control-flow complexity, which accounts for the fact that some of the optimizations

explored do not apply to the majority of the shaders. The few longer shaders with significant control-flow, however, make prime candidates for many optimizations, despite occurring less frequently. The low number of variants for each shader also has the benefit of making it tractable to exhaustively explore the full search-space via iterative compilation.

4.5 Timing Tools and Experimental Setup

In order to explore the performance impact of the compiler optimizations in [Section 4.3](#) on the 147 fragment shaders extracted from GFXBench 4.0 (see [Section 4.4](#)), a custom timing tool was created. The timing tool acted as a micro-benchmark isolating each individual fragment shader's performance characteristics from the rest of the execution environment it was extracted from, and thus allowing smaller changes in performance to be detected without being overshadowed by other computations within its original scene.

This timing tool is described in [Subsection 4.5.1](#). Details of how the corresponding vertex shaders were generated for each fragment shader are discussed in [Subsection 4.5.2](#). Finally, [Subsection 4.5.3](#) explains the hardware used for timing experiments to give the complete picture of the execution environment for the results in [Section 4.6](#).

4.5.1 Shader Execution Environment

To accurately time shaders, they were executed in an isolated context. Injecting them back into GFXBench would cause the performance impact of any single-shader optimizations to be lost in the noise of other shaders and CPU computations. As such, a custom measurement framework was built to repeatedly render full-screen quads using the specified fragment shader, and time the execution of each draw-call.

To reduce the overhead of non-fragment shader stages, only full-screen triangles were drawn. These were clipped to 500*500 pixel quads during rasterization, so only 3 vertex shader calls are required for every 250000 fragment shader invocations. [Figure 4.8](#) shows how a single triangle can be used to span the full visible screen (from -1 to $+1$ in both axes), which is a technique which can be used to optimize performance for any full-screen effect[441]. Performance is gained by invoking the vertex shader the minimum number of times, avoiding any repeated shading of pixels occurring at the seams if 2 or more triangles were used to span the screen, and improving data

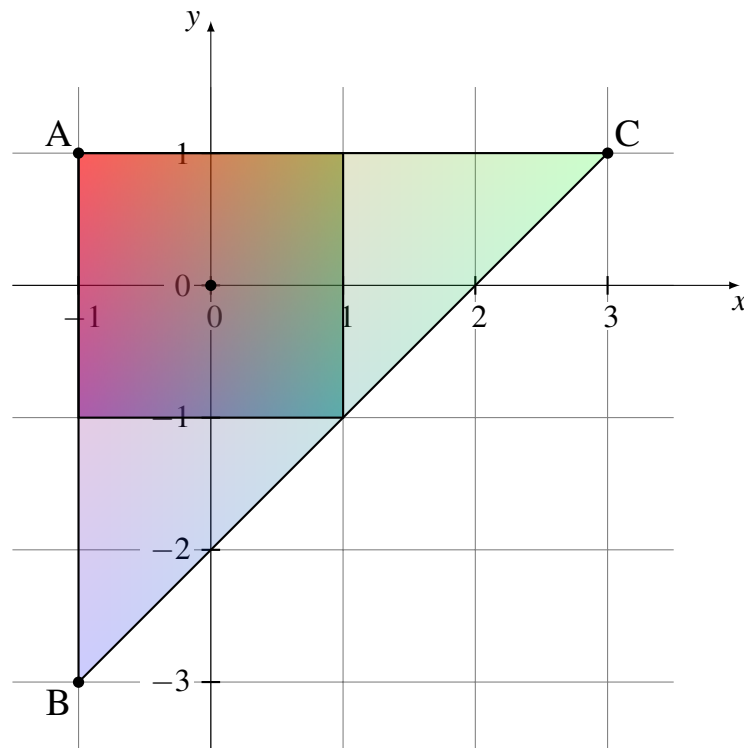


Figure 4.8: The triangle ABC extends beyond the limits of the screen's visible area between -1 and $+1$, and is cropped down to form a full-screen square

locality for texture look-ups and other data retrieval across the screen[442].

Each frame, 1000 triangles (100 on mobile devices) were drawn, and the draw calls were timed using queries to `GL_TIME_ELAPSED`. Although these queries can be noisy and introduce profiling overhead, and better vendor-specific instrumentation may be available, `GL_TIME_ELAPSED` provided a simple cross-vendor comparison metric that was accurate enough for basic performance results. The triangles were drawn back-to-front to ensure that fragment shaders were executed every time, and not elided due to z-buffer comparisons usually used to avoid such overdraw (see [Subsection 3.5.6](#)). The tests were run for 100 frames, and then repeated 5 times per shader variant. These large numbers of samples are used to reduce noise from environmental factors, profiling overhead, and measurement inaccuracies in the timer query API.

As well as the automatically generated minimal vertex shaders described in [Subsection 4.5.2](#), the timing tool also initialises all uniform variables and texture bindings in the target shader. Shader introspection is used to ascertain types and sizes for all uniform inputs, and assigns them automatically to default values (e.g. 0.5 for floats, or a colourfully-patterned opaque power-of-two image for texture bindings). This is not representative of typical shader input, and may circumvent some data-dependent code

paths. More complex techniques such as input fuzzing, or extracting real-world inputs from GFXBench via instrumentation may provide better results, but experiments would take far longer to run due to a combinatorial explosion in input values. As such, the simple approach of using constant inputs was chosen, as it gives a broad overview of performance characteristics without the additional implementation complexity and run-time overhead.

The test harness outputs sequences of draw-call execution times in nanoseconds, which can be compared to determine whether optimizations improve the code. To do this, all time-sequences from a particular shader are combined into a single list, sorted, and then stripped of the top and bottom 5% to remove outliers. Given two sorted lists of times, the normalized delta between pairs of measurements from each list is then taken in order to compare peaks with peaks, and troughs with troughs. Taking the mean of all these normalized deltas gives the percentage speed-up or slow-down for one set of readings compared to the other, which is displayed in the graphs in [Section 4.6](#).

4.5.2 Vertex Shader Generation

To run a fragment shader, a vertex shader with a matching interface is required. As the timing harness described in [Subsection 4.5.1](#) operates by rendering full-screen triangles, the vertex shaders from GFXBench are not suitable, as they may distort the triangle's vertex locations, and could introduce irrelevant overhead. As such, the vertex shaders for the timing harness are automatically generated to ensure regular full-screen triangles are generated with minimal overhead, but the interface of the vertex shader is customized to match the corresponding fragment shader. An example of a generated vertex shader can be seen in [Listing 4.5](#).

In GLSL, the inputs to a fragment shader may be single-precision floating point numbers, integers, and vectors or arrays of these[36]. In GFXBench, only floating-point scalars and vectors are used, so the generation algorithm only considers these types. First, a simple text-based parser detects the name and type of the input variables within the fragment shader. Then, when the vertex shader is generated, output variables of the matching type are inserted and assigned values from several pre-existing computations. Dynamically computed values are used here to ensure no link-time optimization will be able to elide fragment-shader computations associated with these inputs, which would skew any timing results that would optimize them.

```

#version 430 core
uniform float tri_depth;
out gl_PerVertex { vec4 gl_Position; };
vec3 colors[3] = vec3[(
    vec3(1.0, 0.0, 0.0), // Red
    vec3(0.0, 0.0, 1.0), // Blue
    vec3(0.0, 1.0, 0.0) // Green
)];

//Input variables detected from our target fragment shader
out vec2 out_texcoord0;
out vec3 out_normal;
out vec4 out_prevScPos;

void main() {
    vec2 texcoord;
    texcoord.x = (gl_VertexID == 2) ? 2.0 : 0.0;
    texcoord.y = (gl_VertexID == 1) ? 2.0 : 0.0;
    vec4 pos = vec4(texcoord * vec2(2.0, -2.0) + vec2(-1.0, 1.0),
        tri_depth, 1.0);
    gl_Position = pos;

    out_texcoord0 = texcoord;
    out_normal = colors[gl_VertexID];
    out_prevScPos = pos;
}

```

Listing 4.5: Example auto-generated vertex shader for a fragment shader requiring several types of input variables

Each vertex is assigned a unique ID number in the vertex shader invocation, which is guaranteed to be between 0 and 2 for our 3-vertex triangle. This value can then be used to produce different values at each vertex, such as its position, colour, or texture UV coordinates. The values calculated by the example shader in [Listing 4.5](#) are shown in [Table 4.2](#), which also correspond to the shaded triangle shown in [Figure 4.8](#). Selecting the UV coordinates to be between 0 and 2 on the extremities of the triangle in this way ensures that the values will be between 0 and 1 for all the shaded pixels visible on the screen, and therefore index into the valid region of texture image without invoking tiling, clamping, or other context-dependent edge-case texturing operations. As described in [Section 2.2](#), these values calculated at the vertices will be linearly

interpolated across the face of the triangle, which is the cause of the gradient-effect of the colours in [Figure 4.8](#), which are simply blended between the red, green, and blue values at the three vertices.




	A	B	C
Vertex ID	0	1	2
Position (x,y)	(-1, 1)	(-1, -3)	(3, 1)
Texture (u,v)	(0,0)	(0,2)	(2,0)
Colour			

Table 4.2: The colour, position, and texture coordinates at each vertex of triangle *ABC* in [Figure 4.8](#) as calculated by the vertex shader in [Listing 4.5](#)

The vertex *x* and *y* positions are generated as shown in [Table 4.2](#), but the *z* coordinate is assigned using the uniform variable `tri_depth`. This allows the CPU to set a different depth parameter for each triangle it renders, enabling it to simply order the triangles from back-to-front, ensuring every pixel is overwritten with each draw-call. The fourth and final coordinate of `vec4 gl_Position` is assigned the constant 1.0, which is part of a convention ubiquitous within graphics applications[13] known as homogeneous coordinates[443], which allows various 3D transformations to be performed via simple matrix multiplication. As all parts of each vertex’s data are calculated within the vertex shader, no vertex buffer is required.

The values calculated in [Table 4.2](#) are re-used for all vertex shader outputs to avoid unnecessary computational overheads. For all `vec4` outputs, the vertex’s position is used. All `vec3` outputs receive the RGB colour value, which will be between 0.0 and 1.0 for all coordinates, making it suitable for use as a normal vector as well. All `vec2` values are assigned the value of `texcoord`, ensuring they are between 0.0 and 1.0 for all pixels within the visible screen region. Floating-point scalars are assigned with the R value of the RGB colour, and all integer values are assigned with the vertex ID number, or vectors composed of it.

This scheme for generating default values allows for minimal computational overheads, while still ensuring that the computations are dynamic enough not to be constant-folded into the fragment shader. The values selected here are also likely to be within the expected ranges for most variables of each type, so will seldom trigger edge-case behaviours, and tend to execute through typical control-flow paths. Using this scheme, simple vertex shaders were generated for all 147 fragment shaders from GFXBench

4.0, allowing them to be executed within a controlled benchmarking environment designed to emphasise the performance costs of the fragment shader stage for more accurate measurement.

4.5.3 Hardware

Timing experiments were run on 3 PCs and 2 mobile phones, each with a GPU from a different hardware vendor.

These devices are referred to throughout this chapter by their GPU vendor company, but it should be noted that different hardware generations, driver versions, and operating systems may perform differently even for GPUs from the same vendor.

Desktop

The selected desktop platforms were fitted with identical hardware apart from their GPUs. Each had 16GB of RAM, an i7-6700K CPU, and Ubuntu 16.10 installed. The GPUs and drivers chosen for each vendor were as follows:

- **NVIDIA** - GeForce GTX 1080, with OpenGL 4.5 and NVIDIA proprietry driver version 375.39
- **AMD** - RX 480 (8GB), with OpenGL 4.5 and Gallium 0.4 on AMD POLARIS10 (DRM 3.3.0 / 4.8.0-37-generic, LLVM 3.9.1) from Mesa 17.0.0-devel
- **Intel** - HD Graphics 530 (embedded on the i7-6700K), with OpenGL 4.5 and Mesa DRI Intel(R) HD Graphics 530 (Skylake GT2) from Mesa 17.0.0-devel

Mobile

GPU timer queries have been available on desktop since 2009 via `ARB_timer_query` [444], and have been integrated into OpenGL since version 3.3 in 2010[445]. However, they are only available on mobile via the `EXT_disjoint_timer_query` [446] extension. As such, the benchmarking phones were selected to support this extension, allowing the same timing techniques to be used in both mobile and desktop versions.

The mobile hardware selected was an HTC10 (with a Qualcomm GPU), and a Samsung Galaxy S7 (with an ARM GPU). Both ran Android 7.0, and had the following GPUs and CPUs:

- **ARM** - Mali-T880 MP12 (on Exynos 8890 with quad-code Mongoose CPU and quad-core Cortex-A53 CPU)
- **Qualcomm** - Adreno 530 (on Snapdragon 820 with Kryo quad-core CPU)

4.6 Timing Results

This section examines the performance impact of the optimizations from Section 4.3 on the GFXBench 4.0 fragment shaders (see 4.4) across all target platforms (see 4.5.3). The effectiveness and applicability of the optimizations is discussed, along with the performance variability across GPUs.

4.6.1 Overall Performance

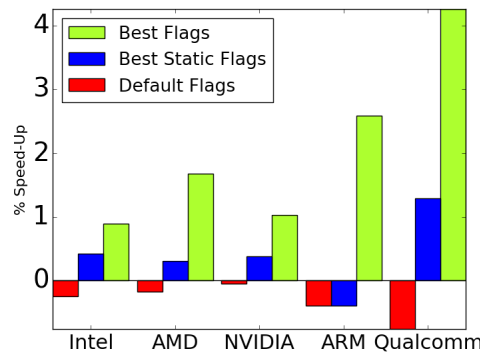


Figure 4.9: Average percentage speed-ups across all shaders.

In Figure 4.9, our technique achieves average speed-ups of 1-4% across all shaders. In contrast, the default LunarGlass transformations give average slow-downs of 0-0.7%.

For some shaders, optimization leads to more substantial gains. The 30 most improved shaders on each platform (Figure 4.10), show average speed-ups of 4-13%. Some shaders experience gains as high as 25% (see Figure 4.11).

4.6.2 Best Static Flags

The "best static" flags in these diagrams are chosen by selecting the flag sequence with the highest average speed-up per platform for all shaders. These flags are shown in Table 4.3, and represent the optimal compilation settings to use if you cannot adapt

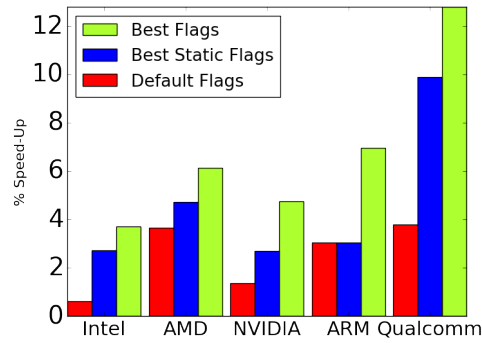


Figure 4.10: Average speed-up for 30 shaders with the highest average per platform.

Platform	Flag							
	ADCE	Coalesce	GVN	Reassociate	Unroll	Hoist	FP Reassociate	Div to Mul
Intel	-	✓	-	-	✓	-	✓	✓
AMD	-	✓	-	-	✓	-	✓	✓
NVIDIA	-	✓	-	-	✓	-	✓	-
ARM	-	✓	✓	✓	✓	✓	-	-
Qualcomm	-	✓	-	-	-	-	✓	✓
All	-	✓	-	-	✓	-	✓	✓

Table 4.3: Best static flags for each platform: Flags that maximise the average speed-up across all the benchmark shaders.

on a per-shader basis. This shows that most platforms share similar flag preferences (ARM being the notable exception).

It is interesting to note that the best flags chosen experimentally are not the flags enabled by default (apart from for ARM). The default GVN, integer reassociation, and hoisting passes are detrimental on average despite being enabled by default. The ADCE pass never changes the output code, so can be safely omitted from the minimal optimal flag selection. Also, the new unsafe floating point passes generally have a positive enough impact to be included in the best static set of flags for all platforms (apart from ARM).

This similarity in optimal flags shows a surprising amount of agreement on which optimizations are beneficial to most vendors. However, [Subsection 4.6.4](#) indicates that although vendors share preferences for the presence or absence of optimizations, the actual performance impact varies.

4.6.3 Per-shader Results

[Figure 4.11](#) shows the performance distributions across all the individual shaders. All

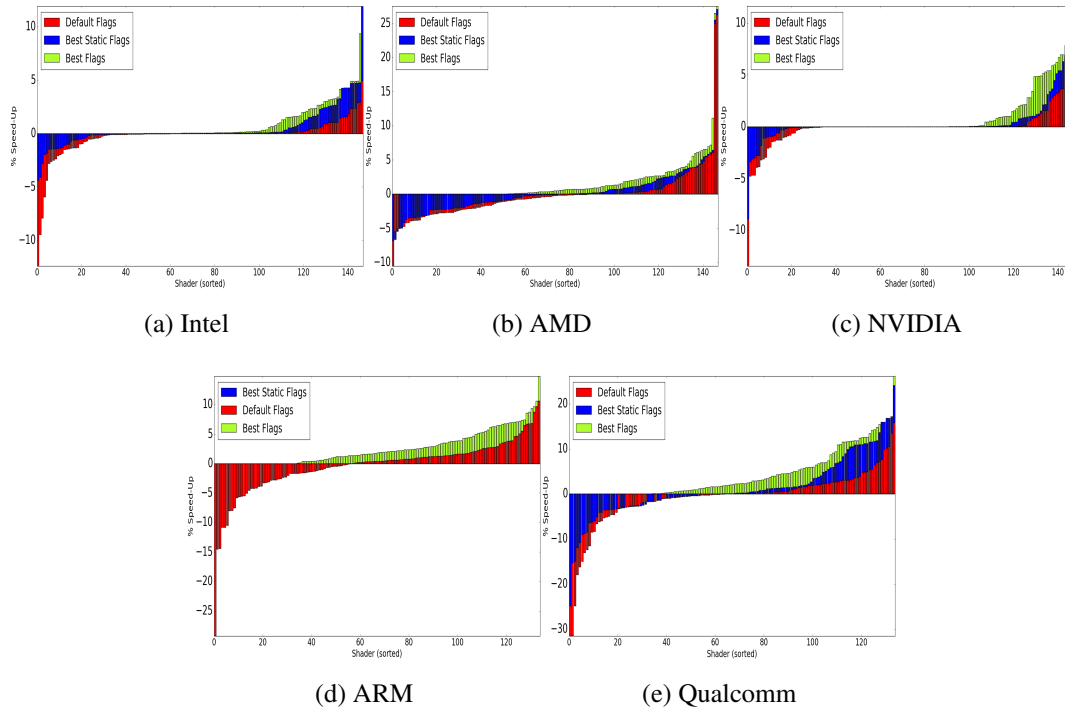


Figure 4.11: Percentage speed-up per shader for each platform. Green shows the best possible performance, red is for default LunarGlass settings, and blue is for the best static set of flags from Table 4.3

graphs have peaks and troughs on either end of a large near-zero mid-section. This demonstrates that frequently, optimization has little effect on shaders, but there are large performance peaks to strive for, and large performance troughs to avoid.

These graphs have large near-zero tails (particularly NVIDIA and Intel), where optimizations have little impact. This is due to the relative simplicity of many shaders in the benchmark suite (see Section 4.4), and the low applicability of many optimizations (see Figure 4.12).

There are also cases where all optimizations cause slow-downs due to source-to-source compilation artefacts (see 4.3.3), or instances where loop unrolling and conditional flattening cause huge basic blocks which can strain register allocation code in the GPU vendor’s compiler.

Despite these negative and near-zero cases, where the optimal strategy is leaving shaders untouched, there are still non-negligible performance gains available for around 25%.

On AMD, the biggest gains are available from some of the default passes such as loop unrolling, so the default LunarGlass results are quite close to the optimal speed-

ups.

On platforms such as Qualcomm and Intel, much of the performance boost comes from the new unsafe floating point reassociation passes, and the default LunarGlass flags are closer to zero in these situations. This results in a larger blue area on the graph, because the main performance gains are from enabling these optimizations for all shaders, so there is less requirement to iteratively tune them.

On ARM and NVIDIA, there are large green areas on the graphs, and small blue ones (the best-static and default LunarGlass settings are the same on ARM). This indicates that there is more to be gained from better flag selection heuristics on these platforms, as a single static set of flags does not guarantee significant performance improvement here.

All the graphs in [Figure 4.11](#) demonstrate that there are both large performance gains and performance pitfalls of between 10-30%. In many cases, the combination of boosting the maximum performance with the new custom passes, and eliminating poor optimizations enables significant improvements over the default LunarGlass results.

4.6.4 Per-Flag Results

Here, each flag's individual applicability and performance impact on each platform is examined. [Figure 4.12](#) shows how frequently each flag applies to a shader, and how often using that flag results in optimal code. Green means it has a positive impact and denotes the number of times where the flag is frequently in the best performing codes, red means it has changed the output code while blue denotes the amount of code unaffected by the transformation

[Figure 4.13](#) shows the performance impacts of each flag when used in isolation. Due to LunarGlass's compilation artefacts (see [4.3.3](#)), LunarGlass running with all optional optimizations disabled is used as a baseline here, rather than the original unaltered shader. This ensures the individual optimization passes' impact is measured, rather than the effect of the code generation artefacts. All the performance violins are centred close to zero due to all the low complexity shaders where the flags either do not affect the code (see [Figure 4.7](#)), or change the source but do not impact the execution speed. As such, the extents and general shape of the violins are more interesting to observe than the mean values.

4.6.4.1 Aggressive Dead Code Elimination (ADCE)

As can be seen from [Subfigure 4.12h](#) ADCE in practice never changes the source output. There is no green region showing it has a positive impact or a red region showing it has any impact. It should result in exactly zero speed-up in the absence of noise, and can likely be safely omitted for most real-world shaders. However, this does not imply that dead code elimination itself has no impact. Trivially dead instructions get removed regardless of which flags are set, so ADCE's lack of effect simply means LLVM's `isTriviallyDead` function (plus extensions for GLSL-specific commands such as `discard`) are sufficient to remove all the dead code.

4.6.4.2 Global Value Numbering (GVN)

GVN applies mainly to the few more complex shaders with many unique optimization variants, and generally has negative but near-zero impact. On Intel, its results are very small, but generally negative. On NVIDIA, its effects are centred around zero, but with one one example of 5% slowdown dragging its average impact down. Qualcomm, on the other hand, experiences gains of around 15% in some cases of using this flag, resulting in its average speed-up being positive. Across all platforms, GVN is in optimal set for less than 50% of the shaders it applies to, so seldom improves code, even in the few cases where it applies.

4.6.4.3 Reassociate

The integer reassociation pass ([Subfigure 4.12c](#)) is rarely applicable, because integers occur very rarely in GLSL shaders. Most cases where it has any impact are actually removing unnecessary additions of zero in floating point calculations, rather than optimizing integer calculations. This pass produces near-zero impact in most cases, and makes things worse in a few, especially on NVIDIA where performance dips by 6% in one case. Integer reassociation almost never occurs in a shader's optimal set of flags, largely because its main use cases are eclipsed by the floating point reassociation pass instead. The low applicability, and chances for slow-down makes this flag one of the least beneficial in LunarGlass.

4.6.4.4 Floating Point Reassociate

By contrast, the floating point reassociation pass applies to $> 50\%$ of shaders, and frequently occurs in their optimal set of flags (see [Subfigure 4.12d](#)). This high applica-

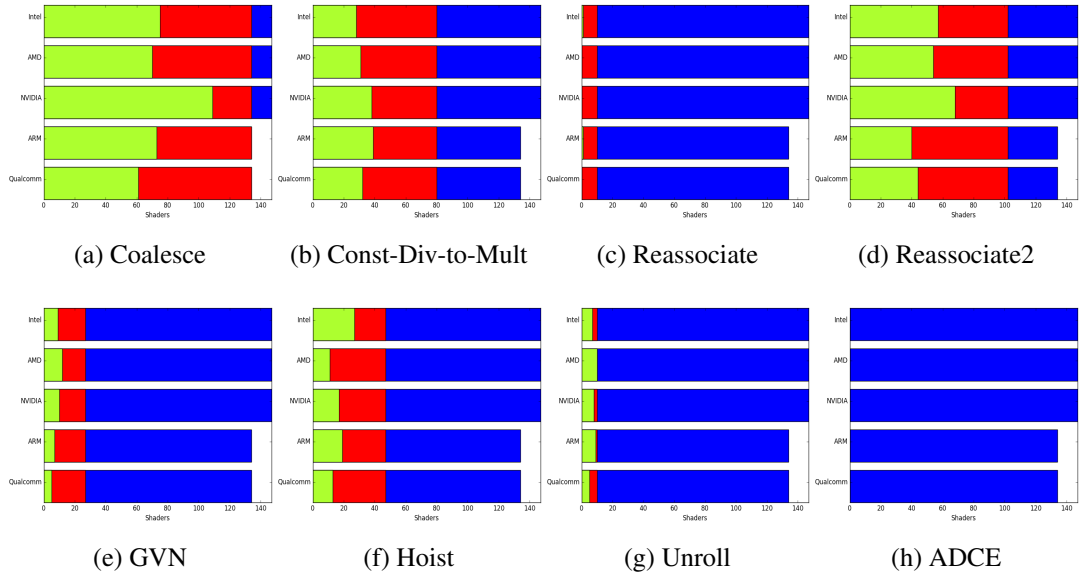


Figure 4.12: Fractions of shaders where optimization passes apply, and have positive impacts. Blue is the total number of shaders. Red is how many the flag has any impact on the output code for. Green is number where the flag is included for at least half of the optimal 10% of variants for that shader.

bility gives this pass a wider spread of values. All platforms except ARM agree on its average positive impact, with peaks of around 5% improvement on desktop platforms, and peaks of around 25% on Qualcomm. However, its results are not universally positive, and despite being Qualcomm’s highest performance peak, it is also its lowest trough at -15%. On ARM, one 20% slow-down drags the average low enough omit it from ARM’s best static flags. The wide spread of results, and the fact this flag appears in shaders’ optimal flag sets around 50% of the time, may indicate that although this pass’s core ideas result in speed-ups, further refinement is needed to reduce slow-down cases. Dividing this pass into smaller components and using better heuristics may achieve the performance gains without all the pitfalls.

4.6.4.5 Loop Unrolling

Loop unrolling (Subfigure 4.12g) is seldom applicable (as few shaders contain loops), but is almost universally positive. On AMD, loop unrolling always improves performance, and can result in 35% gains. On ARM, despite some slow-downs, it reaches a peak of 25%, making it the best flag on ARM as well. On Intel, its effect is near-zero, with a slightly larger slowdown than speed-up. On NVIDIA, it is also near-zero, but

with a peak of 5% improvement. Qualcomm is the only platform where unrolling is not included in its best static flags (see [Table 4.3](#)), and the 8% drop shown in [Subfigure 4.13e](#) may indicate why (although it also achieves gains in some cases too, and hovers near-zero for the most part). For most shaders, unrolling is one of the optimal flags on every vendor, and is a high-impact, low applicability transformation.

4.6.4.6 Hoist

The hoist flag ([Subfigure 4.12f](#)) applies to around 25% of shaders, but is in the optimal set for less than half of them. On most platforms, a single pathologically bad case drags the average down massively. On Intel, it drops 11%, on AMD 7%, on NVIDIA 5%, and on ARM it reaches a massive 35% slowdown. Hoisting sometimes improves all platforms, but these steep pitfalls indicate it should be used with caution, and good heuristics for when to apply it would be valuable.

4.6.4.7 Constant Division to Multiplication

Changing constant division to multiplication (see [Subfigure 4.12b](#)) is possible in >50% of shaders, but is only in the optimal set for around half of these. This is likely an optimization many vendors perform already, so this flag may have little impact. It might be a coin-toss whether results are negative or positive (hinted at by its symmetrical results in [Figure 4.13](#)), and could occur in optimal sets because it has a near-zero impact, so can be toggled on or off safely without slowing down shaders. On Intel (which has the least measurement noise), its impact is almost zero in all cases. The results for NVIDIA, AMD, and ARM are symmetrical and centred around zero, with ranges around 4%, 10%, and 10% respectively. However, Qualcomm's results range from +25% to -13%. This pass's wide applicability makes it difficult to tell whether the graphs show genuine improvements, or merely each platform's measurement noise (which the symmetrical results might back up).

4.6.4.8 Coalesce

The coalesce flag applies to almost every shader (see [Subfigure 4.12a](#)) because they frequently insert elements into vectors. Its results span a wide range, with most averages near-zero, or slightly negative impact (see Qualcomm). However, this is at odds with its inclusion in the best static flags in [Table 4.3](#), so the largely symmetrical spread of results may be due to measurement noise again. Occurring in optimal sets for so

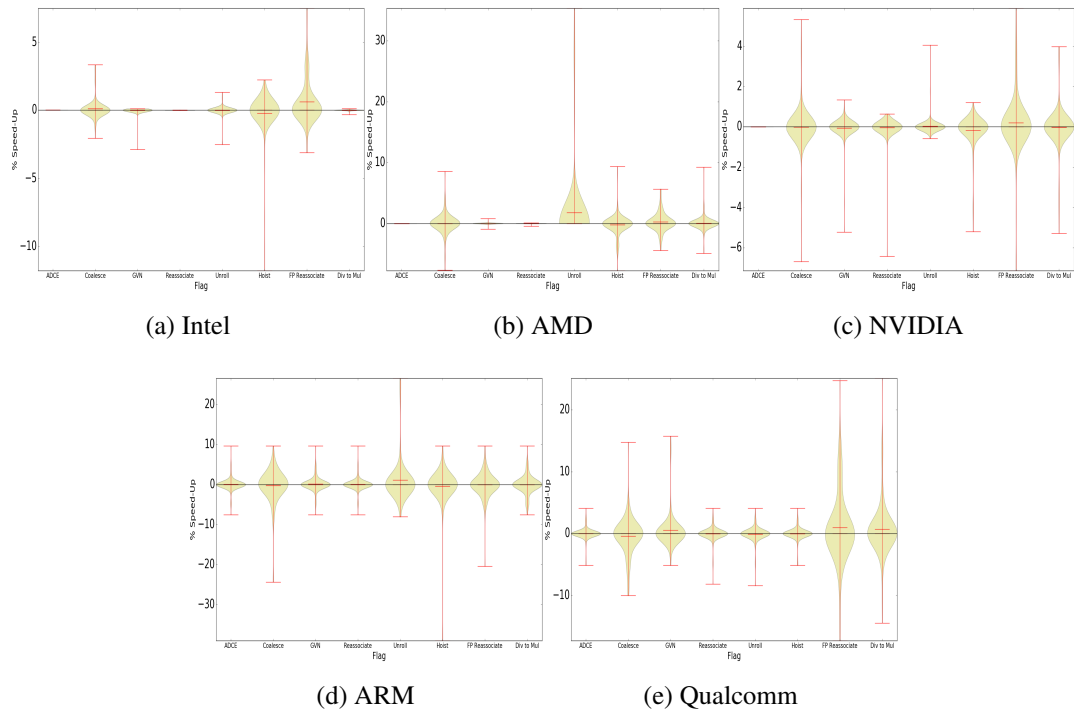


Figure 4.13: Percentage speed-up from individual flags for each platform

many shaders shows it is frequently favourable to include, although different platforms have slightly different preferences. NVIDIA prefers it almost always enabled, but on Qualcomm it is optimal for around 50%, so is less critical.

4.6.5 Summary

Only floating point reassociation, loop unrolling, and hoisting have sufficiently large impacts to affect shaders in the absence of other passes. Also, as there are few distinct variants for each shader (see [Figure 4.7](#)), the optimal 10% of variants is often only a single shader, so the optimality in [Figure 4.12](#) may also be somewhat fickle. However, the larger visible performance trends, and the number of shaders each type of optimization pass applies to, gives some interesting insight into the nature of graphics shaders in general, and how frequently different optimization opportunities arise.

4.7 Conclusion

This chapter explored the impact of common compiler optimizations on fragment shaders across 3 desktop and 2 mobile GPUs. Shaders were extracted from the popu-

lar graphics benchmark GFXBench 4.0[8], and aspects of their complexity and typical algorithms were characterized. The offline source-to-source compiler LunarGlass[10] was then used to transform these fragment shaders using different combinations of optimization techniques. A timing tool was developed to isolate the fragment shader execution performance by timing the repeated rendering of full-screen triangles. This tool allowed comparisons between different optimization strategies via iterative compilation, enabling the optimal set of optimization passes to be determined for each shader.

The work here demonstrates that although shaders undergo vendor-specific compilation, offline source-to-source optimizations can still have significant positive and negative impacts, which vary across optimizations, benchmarks and platforms.

Chapter 5

Analysis of Potential Optimizations Within Shader Pipelines

5.1 Introduction

In [Chapter 4](#), the optimizations explored focused on individual fragment shaders extracted from a single GPU benchmark suite. In this chapter, execution traces are extracted from commercial PC games, and static analysis of the shaders and the data flowing into them is performed. This extends the scope of potential optimizations beyond individual shader pipeline stages, and explores data flowing between the different execution stages, and also between the CPU and GPU.

[Section 5.2](#) begins with motivating examples of the different optimizations considered throughout this chapter, which are explained in more detail in [Section 5.3](#). The trace analysis and static analysis techniques performed on shaders to detect these optimization opportunities are explained in [Section 5.4](#). The games used as benchmarks are discussed in [Section 5.5](#), with the analysis results on the shaders presented in [Section 5.6](#). Analysis of data throughout the whole execution trace given in [Section 5.8](#), along with some performance results from manually prototyping some of the optimizations to remove dead data.

The overall aim of the work in this chapter is to explore real-world data from commercial games, detect potential areas for optimizations, and measure how prevalent these opportunities are. This work is then extended in [Chapter 6](#) with implementations to exploit and measure the performance benefits of some of the opportunities discovered via the analyses performed in this chapter.

5.2 Motivating Example

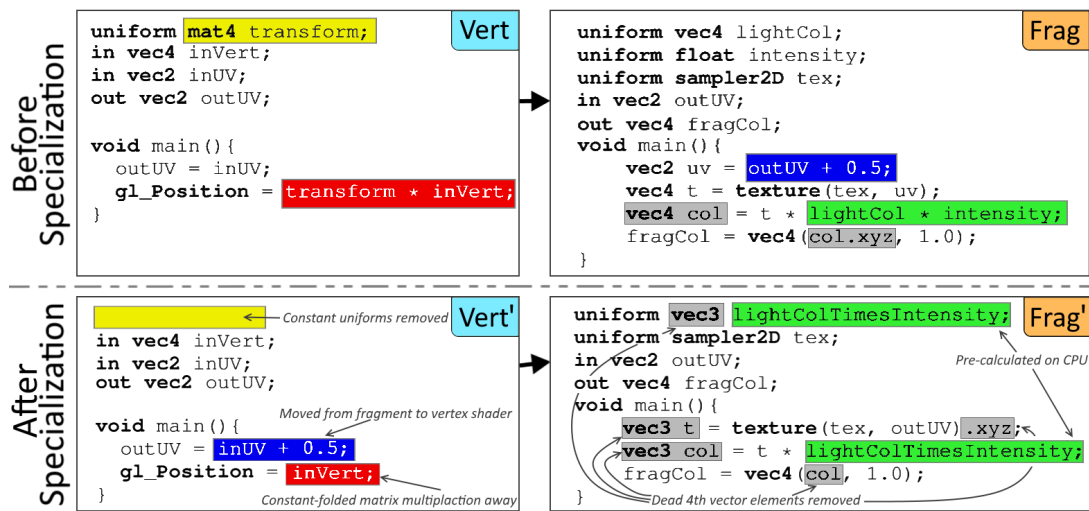


Figure 5.1: Example vertex + fragment shader pipeline before and after code specialization. Coloured highlights in the "before" pipeline show specialization opportunities. After specialization, these highlight what has changed. `transform` is known to be the identity matrix through run-time profiling.

- Yellow box - constant
- Red box - constant-foldable
- Green box - movable to CPU
- Blue box - movable to vertex shader
- Grey box - contains dead elements


This section will demonstrate the different types of specialization and code motion techniques examined throughout this chapter via the example code in Figure 5.1, which depicts a simple shader pipeline before and after several specialization and optimization techniques were applied. The top two boxes Vert and Frag show the vertex and fragment shader code before applying any specialization. The arrow between the boxes denotes that the variable `outUV` is an output from the vertex shader getting passed as an input to the fragment shader. The lower two boxes Vert' and Frag' refer to the code after specialization has taken place.


In Vert and Frag, a number of opportunities for specialization are highlighted with colour-coded boxes. In Vert' and Frag' the highlights show the effects of the specialization. The specialization types examined are as follows:


Constant Uniforms. Consider the upper left hand box Vert and the declaration of `transform` highlighted in yellow. Run-time profiling has shown that `transform` is constant, and is known to be the identity matrix. This means that the declaration can be removed after specialization as shown in yellow in the lower program Vert'.

Constant-Folding - Knowing that `transform` is constant and the identity ma-

trix lets us avoid the unnecessary matrix-vector multiplication `transform * inVert` highlighted in red in `Vert`. This multiplication is removed after specialization to give just `inVert` as highlighted in red in the vertex shader `Vert'`.

 **GPU-CPU Code Motion** - Calculations using only uniforms can be pre-computed on CPU. The calculation `lightCol * intensity` highlighted in green in the `Frag` code of [Figure 5.1](#) is an example of this. In the transformed code of `inFrag'`, this multiplication is performed on the CPU, and is passed in via the new uniform `lightColTimesIntensity`.





 **Fragment-Vertex Code Motion** - Consider the calculation `outUV + 0.5` highlighted in blue in the fragment shader `Frag`. Its value is unaffected by linear interpolation, so can be moved to the vertex shader `Vert'` also shown in blue. This lets the fragment shader use the raw value of `outUV` for texture look-ups, potentially enabling better pre-fetching.

 **Removing Dead Vector Elements** - Considering again the code in `Frag`, only 3 components of `col` are used for the fragment shader's output, highlighted in grey. Knowing this, the 4th elements of `t` and `lightCol` can also be determined to be also unused. After specialization, all these variables can be changed to `vec3s` instead, potentially saving registers, uniform buffer space, and texture unit bandwidth as shown in `Frag'`.

5.3 Example Optimizations

This section provides more detail about the specialization ideas demonstrated in [Section 5.2](#), and explains when they occur and how they might benefit performance. First, several terms used throughout the chapter will be defined:




Specialization Type: This chapter focuses on the following specialization opportunities:

-  *Dead* code or data - unused and can be removed
-  *Movable* code - can be transplanted to a different location
-  *Constant* data - knowable at compile time
-  *Constant-foldable* code - computable at compile-time

Code Location: Code can be executed in the following locations:

- *CPU* - Executed beforehand, and passed to shaders via uniform variables
- *Vertex Shader* - Executed in parallel on GPU as vertex shaders
- *Fragment Shader* - Executed in parallel on the GPU as fragment shaders

Data Location: Data can come from the following locations:

-  *Uniforms*; - The same value is passed to all shader instances from the CPU
-  *Inputs* - from a prior pipeline stage. Fragment shaders receive these from the vertex shader, which receives its inputs from the vertex buffer
-  *Textures* - Read from 2D images

Analysis Type: The following types of analysis are performed:

- *Static* - examining only shader source-code
- *Oracle* - an estimated upper bound on specializations
- *Dynamic* - using online profiling or offline trace analysis to determine data values

Specialization Granularity: Every element of vectors, matrices, and arrays is regarded individually. Specializations may be applied to variables of these aggregate types either:

- *Fully* - all elements can be specialized
- *Partially* - only some elements can be specialized, but could be extracted

Now that all the terminology has been defined, the different optimizations will be discussed in more detail.

5.3.1 Dead

Dead code and data elements in shaders can be detected via static source-code analysis (see [Subsection 5.4.1](#)), and provide the following optimization opportunities:

Dead Code: When some elements of an instruction's output have no impact on the final result. These may be pruned to reduce calculations, free up registers, or allow other non-dead elements to occupy their place.

Dead code can be eliminated using only static data. The driver's shader compiler may cull fully dead variables, but pruning partially dead ones is less common. Element-wise pruning may allow more compact storage e.g. combining two half-dead `vec4`s into a single live `vec4`. However, it must be applied carefully, as it may introduce unnecessary `mov` instructions or remove useful padding.

Dead Data: Elements in a shader's uniform, input, or texture interface that are either never loaded from, or the loaded result is never used. Removing dead data from a shader's interface may improve cache usage, and reduce the amount of memory, communication, interpolation slots, or texture look-ups required.

Eliminating dead data requires more contextual knowledge. Pruning dead uniform data alters the CPU-GPU interface, and may cause alignment issues, but can reduce communication and improve caching. Exploiting dead vertex shader inputs also alters the CPU-GPU interface, but can significantly lower the memory footprint and input assembly time for meshes.

Eliminating dead fragment shader inputs requires shader linkage knowledge, as both the vertex and fragment shaders' interfaces are altered. This can reduce vertex shader computation and use fewer interpolation slots. Dead texture look-ups can be culled using only static data. OpenGL drivers perform some of these optimizations already, such as culling fully dead uniforms, but greater gains are possible by altering interfaces and exploiting partially dead data.

Most OpenGL drivers will already prune some fully dead uniform data, and allow users to query whether uniforms are active before storing data to them. However, exploiting partially dead data might allow even more compact uniform buffers for reduced communication and better caching. Re-packing uniform data requires altering the CPU-GPU interface, and care must be taken to avoid alignment issues.

The GPU driver's linker may eliminate some fully dead fragment shader inputs, but is less likely to exploit partially dead variables. Doing so requires knowledge of which pairs of vertex and fragment will be linked together, as the interface must be altered on

both ends. Partially dead vertex shader inputs can also be pruned if all shaders using that vertex buffer are known, and the CPU code can be altered to store the data more compactly. This may significantly reduce the memory footprint of meshes, and lower input assembly time.

Fully dead texture loads are culled via dead code elimination, and partially dead loads can be optimized statically by using GLSL's swizzle operator on the `texture` call to load only some elements. More advanced specializations may involve combining partially dead textures together, or avoiding loading and binding of dead textures from the CPU.

5.3.2 ■ ■ Movable

As noted by many prior researchers (see [Subsection 3.3.4](#)), performance gains may be achieved by reducing the rate of certain calculations. Much of that prior work involved performance-accuracy trade-offs caused by such rate reductions, but there are also certain calculations that may be moved elsewhere in the graphics pipeline without incurring additional accuracy trade-offs. Static analysis can be used to detect areas of shader code which may be transferred to different pipeline stages or to the CPU, which will typically reduce the rate at which such computations occur.

■ **CPU-Movable:** Some code can be moved from GPU shaders to the CPU. This can shrink the shader's uniform interface if the pre-computed results are smaller than the components of the calculation. Pre-computing values on the CPU may also improve performance if the GPU is the bottleneck. If branch conditions are evaluated on the CPU, it can select different specialized shader pipelines without GPU branches.

■ **Vertex-Movable:** If calculations are unaffected by linear interpolation, they can be transferred from the fragment to the vertex shader, which is typically invoked less often as objects usually have fewer vertices than pixels. This can reduce the total GPU calculations, use fewer vertex-fragment interpolation slots, and eliminate dynamically indexed texture look-ups. This code motion alters the vertex-fragment interface, and may be detrimental for meshes with many sub-pixel triangles.

Code motion requires knowledge of either the CPU-side code, or the shader pipeline linkage, as these interfaces must be altered when code is moved. It will not always result in improved performance, so care should be taken that code movement only occurs where it will be beneficial. Using dynamic analysis increases the amount of code motion opportunities beyond those visible using only static data.

5.3.3 Constant

Detecting data that is constant at run-time enables further optimizations and specializations. In addition to the coloured squares used to label specialization opportunities, coloured circles are used to represent which type of data is constant:

 **Uniforms**  **Inputs**  **Textures**

Constant Uniforms: Uniforms which always take the same value at run-time. Uniforms are stored on a per shader-pipeline basis, and are frequently updated, but are often set to the same value each frame. However, many uniforms remain unchanged throughout an application's entire lifetime, especially when viewed on a per-element basis, so may offer many specialization opportunities.

Constant Inputs: Vertex shader inputs come from the vertex buffer. They are constant if they have the same value for all vertices in all buffers the pipeline uses (which is fairly rare). Fragment shader inputs come from vertex shader outputs, which may be constant at compile-time, or via propagating dynamic data detected at run-time.

Constant Textures: If a texture's colour channel is the same for every pixel, elements read from this channel are constant. This is common for fully opaque textures with constant alpha channels of 1.0. Textures representing a square that is a single colour are another example where all colour channels are constant.

5.3.4 Constant Foldable

Constant-Foldable: Elements that are compile-time constant or known via prior run-time analysis can be folded in the compiler to reduce computation. Folding these constant values into further calculations than those immediately detectable as constant enables constant data to propagate throughout the entire shader, potentially unlocking further optimizations of every kind. For vector data that is partially constant, it may be possible to perform constant folding on only a constant subset of that vector for any component-wise calculations, and then recombining the resulting elements. Such partial propagation may not always be desirable, as it may introduce extra data movement instructions, but the positive effects of computation reduction, and any further optimization steps that are enabled as a knock-on effect may be sufficiently beneficial that even partial constant propagation may be a desirable option.

5.4 Techniques for Detecting Potential Optimizations

This section describes the analyses used to detect the specialization opportunities explained in [Section 5.3](#). The resulting static specialization opportunities detected with these techniques are described in [Section 5.6](#), and the run-time trace analysis of constant uniform data is found in [Section 5.8](#).

5.4.1 Dead Code/Data Analysis

To detect dead elements in the shader code (see [Subsection 5.3.1](#)), a backwards-propagating dataflow analysis can be used. This starts with only final store calls live, and propagates this liveness into every other element used to calculate the existing live ones. After all the live elements were determined, all others were defined to be dead as can be seen in [Algorithm 1](#).

Algorithm 1 Dead Element Detection

```

for all Inst in Instructions do
    if Inst is a store, a terminator, or has side-effects then
        Set all elements of  $Liveness_{Inst}$  to Live
        Add Inst to WorkList
while WorkList not empty do
    Pop Inst from WorkList
    for all Op in operands of Inst do
        if  $Inst \in \{ \text{extract, insert, swizzle} \}$  then
            Set  $Liveness_{Op}$  as a permutation of  $Liveness_{Inst}$ 
        if Inst is an elementwise operation (e.g. add, multiply) then
             $Liveness_{Op} \leftarrow Liveness_{Inst}$ 
        if Inst uses all elements of Op (e.g. dot, normalize) then
            Set  $Liveness_{Op}$  to full
        if Any elements of  $Liveness_{Op}$  changed then
            Add Op to WorkList
    All elements not set to Live by now must be Dead

```





Dead uniform and input data is determined by examining which elements of load instructions are live, and merging results for any loads from the same address. This let

us see not only which variables were declared but never loaded from, but also which ones were loaded from but only partially used.

5.4.2 Movable & Constant Code Detection

To statically analyse which code elements were movable to the CPU or the vertex shader (see [Subsection 5.3.2](#)), a forward-propagating dataflow algorithm can be used. This algorithm also tags which sources different inputs were loaded from, enabling runtime-constant inputs to be explored via an oracle study in [Section 5.7](#).

Every instruction in each shader is iterated through, and every element of their return values are tagged one of the following values if applicable:


-  **U** - Loaded from a Uniform
-  **V** - Loaded as Input from a previous shader stage
-  **T** - Loaded from a Texture
-  **C** - Constant

The following rules can then be used to tag data as constant-foldable or movable:

-  **CC** - Constant-foldable


$$\mathbf{CC} = f(\mathbf{C}, \mathbf{C})$$

where **CC** is the result of an arbitrary function $f()$ whose arguments are all constant.

-  **UU** - Movable from GPU to CPU

$$\mathbf{UU} = f(\mathbf{U}, \{\mathbf{C}|\mathbf{U}|\mathbf{CC}|\mathbf{UU}\})$$

where **UU** is the result of a function $f()$ whose arguments consist of only uniform, constant, constant-foldable, or CPU-movable values.

-  **VV** - Movable from fragment to vertex shader

$$\begin{aligned} \mathbf{VV} = & \{\mathbf{V}|\mathbf{VV}\} \\ & * \{1|\mathbf{C}|\mathbf{U}|\mathbf{CC}|\mathbf{UU}\} \\ & + \{0|\mathbf{C}|\mathbf{U}|\mathbf{V}|\mathbf{CC}|\mathbf{UU}|\mathbf{VV}\} \end{aligned}$$

where the value of \mathbf{VV} will be unaltered by linear interpolation between corners of the triangle, so must be a linear combination of a vertex input and either constants or uniforms.

All these tags are applied to scalar elements individually, and are permuted whenever a vector's elements are inserted, extracted, or shuffled. This gives a fine-level analysis, where the same vector may have different elements that are dead, movable, and constant.

These analysis passes were implemented within LunarGlass[10], an LLVM-based compiler[427] which handles GLSL[36] shaders, which was also used in Chapter 4. The custom liveness and movability analysis passes described in Subsection 5.4.1 and Subsection 5.4.2 were executed after LunarGlass had performed various optimizations on the source-code such as conditional flattening and loop unrolling. A description of all the default LunarGlass optimization passes is provided in Subsection 4.3.1. These passes resulted in code with fewer branches, with most shaders ending up with a single basic block. This improves the applicability of the per-instruction propagation algorithms described here which do not account for dynamic branching and control-flow cycles. This also ensures that the specialization opportunities measured here do not disappear after traditional optimizations such as dead-code-elimination.

5.4.3 Dynamic Trace Analysis

To extract shaders and analyse the run-time behavior of our benchmarks, a modified version of apitrace[12] was used. Widely used for debugging graphics drivers and games, apitrace is an open-source tool that injects itself into applications, and traces all OpenGL API calls. This trace can then be played back and inspected on different devices. Because OpenGL drivers require shader source code to be submitted for compilation, these shaders are recorded as arguments by apitrace. These shaders were extracted from the trace files, and used for static analysis in Section 5.6. This is similar to some of the other library-injection based techniques used by other researchers as described in Subsection 3.5.3, some of which use the same apitrace tool[368].

As well as extracting shaders, custom apitrace passes were also implemented to analyse which of each shader's uniform variables were updated, and which remained constant, as well as which calls to update these values were redundant. This required tracking which shader source strings were bound to which program objects, and which locations were assigned to each uniform, and which value was assigned to each uni-

form location. By tracking the values of all uniform elements individually, including individual scalar elements of vector uniforms, it was possible to perform a fine-grained analysis of which data was constant and which values were repeatedly assigned the same values in redundant calls. In [Chapter 6](#), the tracking and exploitation of constant uniform values is the primary focus, so please see [Section 6.3](#) for a detailed description of the implementation of the tools developed to track this uniform data.

5.5 Benchmark Games

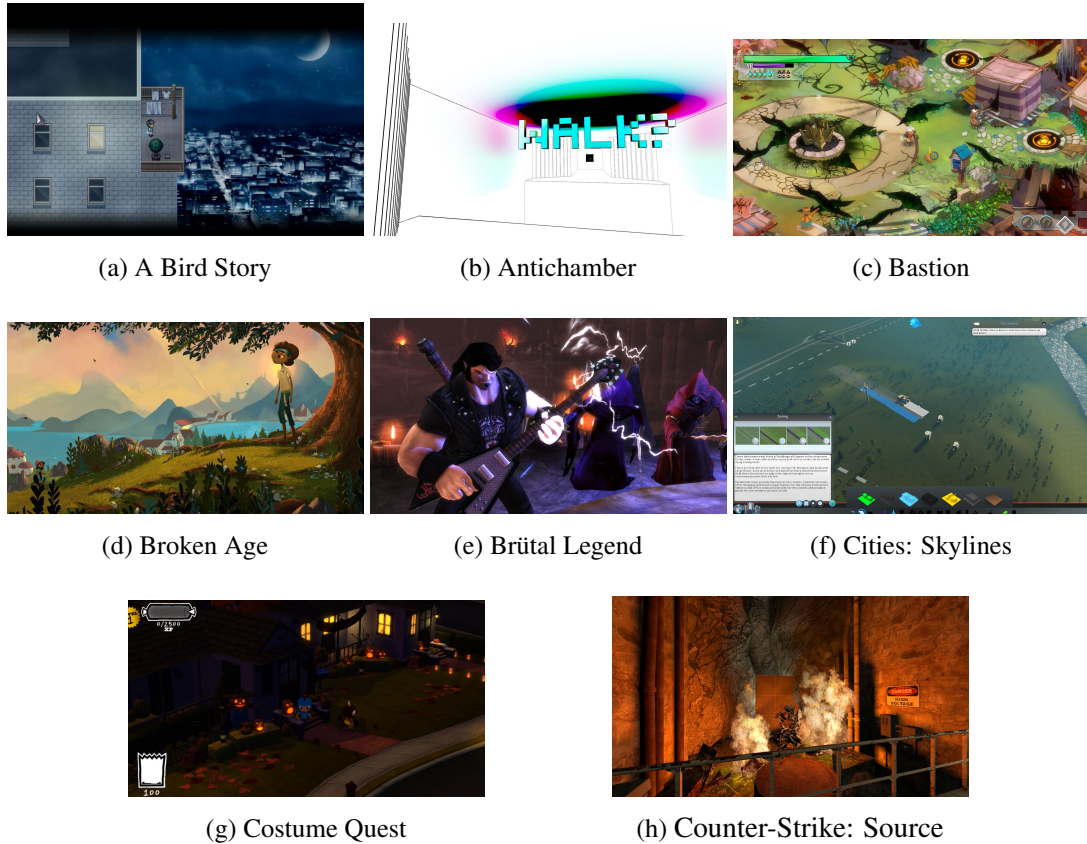


Figure 5.2: Screenshots from the 8 Ubuntu-compatible games selected as benchmarks, covering a variety of different game-engines and art-styles.

A variety of 2D and 3D Linux-compatible games were used to gauge the behavior of typical graphics workloads (screenshots in [Figure 5.2](#)). The table in [Figure 5.3](#) shows the games used, and their total vertex and fragment shaders to give a rough idea of their complexity. As explained in [Section 2.3](#), most PC games use only the Windows-specific DirectX API. As the toolchain developed here uses OpenGL, Linux-compatible games from before the release of the Vulkan API were selected to ensure

the OpenGL was used within. They represent a cross-section of varying complexity 2D and 3D titles from different engines and games studios.

Antichamber uses the Unreal engine[58], and Cities: Skylines uses Unity[59], these being the most popular commercial game engines. Counter-Strike: Source uses Valve's famous Source engine, which is also available for studios to license. Brütal Legend and Costume Quest use the in-house Buddha engine, but have different art-styles. Including both allows us to see whether engines or art-styles have more impact on shaders' data patterns. Bastion uses another in-house engine, but with 2D isometric graphics. A Bird Story uses a modified version of the RPG-Maker, and its simple tile-based 2D graphics barely utilize shaders. Broken Age is a more complex 2D game in the open-source Moai engine, using skeletal animations, parallax layers, particles, and some 3D effects. These games span a wide variety of engines and art-styles of varying complexities, and are a good cross-section of games available on Linux.

Each game used shaders quite differently. "A Bird Story", a simple tile-based 2D game had 12 different shaders, whereas "Antichamber", a minimalistic 3D game using the heavy-duty Unreal Engine had 3000 shaders.

2D games had a higher percentage of shader instructions as texture look-ups. 3D games had more instructions in general, and had higher absolute numbers of texture look-ups, but were proportionally lower than the 2D ones due to the larger amounts of other more complex lighting calculations.

Short Name	Full Name	Engine	2D/3D	Vertex Shaders	Fragment Shaders
Bird	A Bird Story	RPG Maker	2D	16	16
Anti	Antichamber	Unreal	3D	1277	2833
Bast	Bastion	Custom	2D	7	22
BAge	Broken Age	Moai	2D	88	70
Brut	Brütal Legend	Buddha	3D	262	1238
City	Cities: Skylines	Unity	3D	484	484
CQuest	Costume Quest	Buddha	3D	150	888
CS:S	Counter-Strike: Source	Source	3D	3025	503

Figure 5.3: Benchmark games with their abbreviations and numbers of shaders.

5.6 Static Analysis Results

This section presents the results of the shader analysis described in [Section 5.4](#), and shows how many of the specialization opportunities from [Section 5.3](#) occur in shaders from typical games. First, static analysis is used to determine that large percentages of code and data are dead. In [Section 5.9](#), it is shown that removing this dead data can give up to 6x performance improvements in some cases. Further analysis indicates that there are small statically available specialization opportunities in shader code, especially in hoisting conditionals from the GPU to the CPU to avoid branching.

[Section 5.7](#) contains an oracle study to determine the upper bounds of code specializability if all uniform, input, or texture variables were known constants at compile time. The results indicate that large code reductions are possible, especially since many condition variables for branch instructions become constant-foldable.

Based on the results of this oracle study, [Section 5.8](#) examines the actual dynamic values of uniforms and shows that there is both a large amount of constant values and redundant update operations.

5.6.1 Static Dead Code and Data

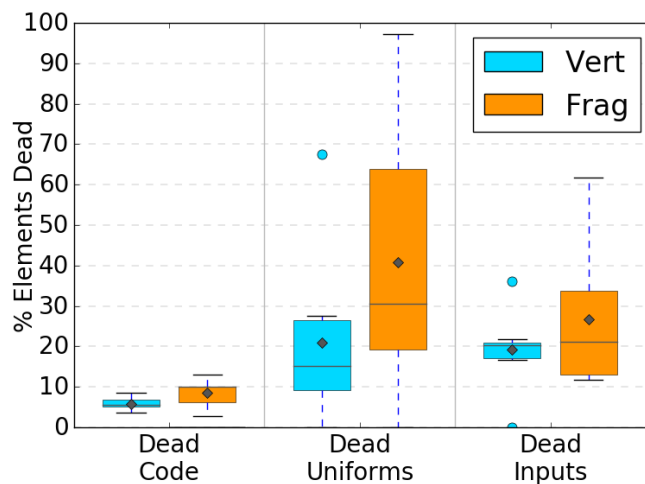


Figure 5.4: % Dead code, and dead uniform and input data. Small portions of code elements, but sizeable amounts of uniform and input data are dead.

Dead Code

[Figure 5.4](#) shows that up to 13% of all scalar elements across all shaders in a game can be statically classified as dead code, with 6 / 9% dead on average for vertex/frag-

ment shaders. This means most games have many shaders with a modest amount of dead code amenable to pruning. This modest amount is not surprising as all fully dead instructions are already stripped out by a previous compiler pass. This proportion remains consistent between games, indicating that many different shaders have opportunities for exploiting partially dead vector elements.

Dead Uniform Data

While there is little dead code, there is significant dead data. On average, 21 / 41% of vertex/fragment uniforms are dead, as shown in Figure 5.4, but there is high variability among games. Some games' uniform interfaces are far larger than necessary. Complex 3D games with automatically generated shaders such as Antichamber, which uses the Unreal Engine[58] can produce pathological cases. Figure 5.5 provides more detail on a per game basis, and shows that 68 / 97% of Antichamber's declared vertex/fragment uniforms are statically dead. The fragment shaders in Counter-Strike: Source also exhibit high proportions of dead uniform data. In contrast, simpler 2D games such as A Bird Story utilize almost every uniform. Overall, fragment shader uniforms contain dead data more often than vertex shader uniforms, and their usage varies more between games. Timing tests in Section 5.9 demonstrate how significant speed-ups can be achieved by exploiting this dead data.

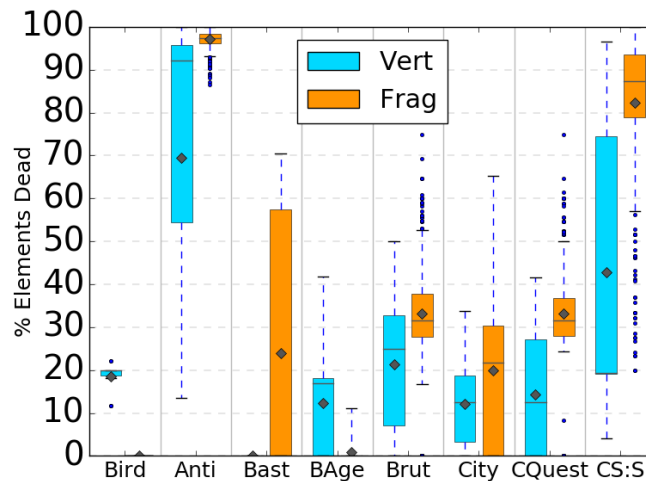


Figure 5.5: % Dead uniform elements for all individual shaders in each game, exhibiting wide variability. Simple 2D games have fewer dead elements than larger 3D games such as Antichamber and Counter Strike: Source.

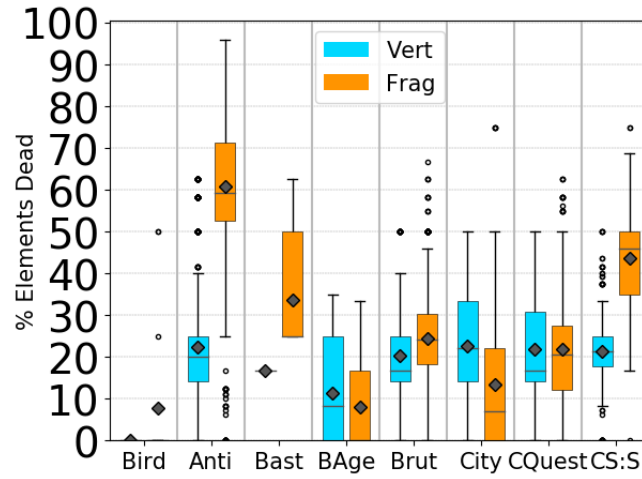


Figure 5.6: % Dead input elements for all individual shaders in each game. Most vertex shaders have $\sim 20\%$ dead inputs. Fragment shaders vary more; simple 2D games show better utilization than complex 3D ones.

Dead Input Data

Input variables are also frequently dead. [Figure 5.4](#) shows that on average, 19% of inputs from the vertex buffer and 27 % from the vertex-fragment interface are dead. There is again considerable variation across games as shown in [Figure 5.6](#). Here 26% of Cities: Skylines vertex inputs and 62% of Antichamber fragment inputs are dead. This means a significant portion of the vertex buffer could be reduced, lowering the mesh’s memory footprint and speeding up the vertex input assembly stage before the pipeline begins.

Inputs from the vertex-fragment interface contain even more dead elements, with an average of 27%, and up to 62% for Antichamber. If the dead data information were back-propagated across the vertex-fragment interface, then vertex computations could be significantly reduced. This could be achieved by modifying the liveness algorithm in [Subsection 5.4.1](#) to tag all store instructions in a vertex shader with the liveness information exported from prior liveness analysis of the corresponding fragment shader in the program pipeline. As well as increasing the proportion of code with partially dead vector elements to be exploited, this analysis could also shrink the size of the interface to lower the number of vertex-fragment interpolation slots and load instructions required.

In order to propagate liveness information between shader stages, it is necessary to determine which vertex and fragment shaders are linked together. Trace analysis such as that described in [Subsection 6.3.2](#) may be sufficient to determine a unique

shader pair that are bound together. In the case where a vertex shader is reused with multiple different fragment shaders, the liveness results for all bound fragment shaders would need to be merged, and the interfaces for all of them would need to be modified. As such, optimizations propagating liveness information in this way are likely best suited as internal link-time optimizations within the driver's compiler, or on a shader source-code level in rendering engines where unique vertex/fragment shader pairs can be determined.

To exploit dead vertex buffer-inputs in the vertex shaders, liveness information needs to be propagated to the CPU-side, as it is the CPU-side OpenGL API calls which determine the vertex-buffer's contents. When stripping dead data from a vertex buffer, care needs to be taken with data alignment, as this can greatly impact the performance of the input assembly stage and subsequent data loads.

5.6.2 Statically Movable Code

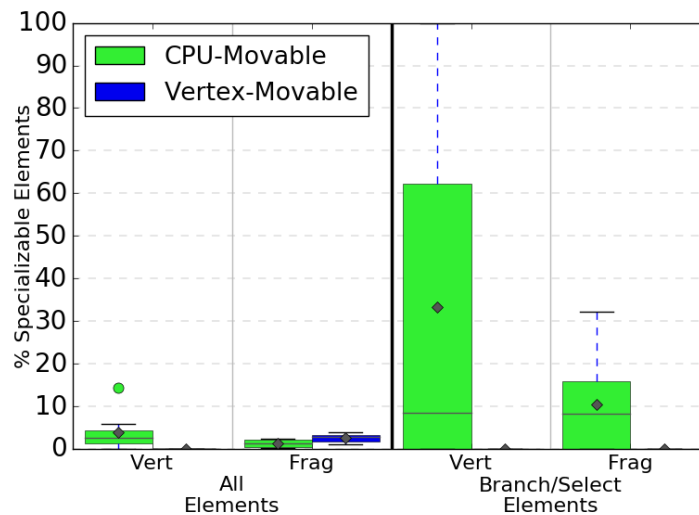


Figure 5.7: % statically movable code for all vertex/fragment shaders - a small percentage of all instructions, but many branching could be hoisted to the CPU.

As well as finding dead code elements, static analysis can also determine how much code is movable to either the CPU, or the vertex shader. [Figure 5.7](#) shows that statically movable code makes up an even lower percentage of shader instructions than dead code. On average, 4 / 1% is movable to the CPU for vertex/fragment shaders, with up to 14% CPU-movable in A Bird Story's vertex shaders (see [Subfigure 5.8a](#)). An average of 3% (max 4%) is movable from fragment to vertex shaders too. This indicates that statically detecting movable code does little to reduce the overall percentage

of instructions.

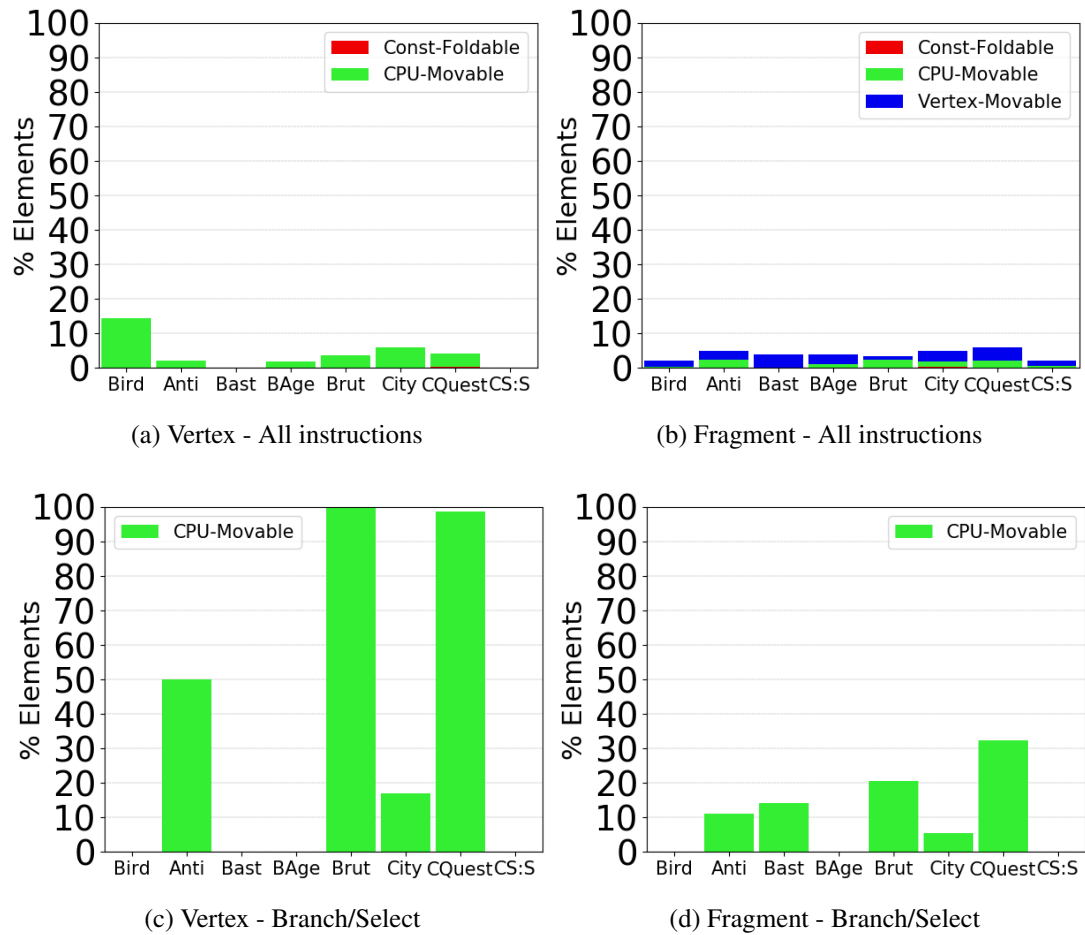


Figure 5.8: Percentage of statically specializable shader instructions across all games.

Branches

Although CPU-movable code forms a small overall proportion of shaders, it is significantly over-represented in the condition variables for branch and select statements, as shown in the right-hand part of [Figure 5.7](#). In fragment shaders, an average of 10% of variables guarding branches or selects can be moved to the CPU, with those in Costume Quest up to 32% CPU-movable (see [Subfigure 5.8d](#)).

For vertex shaders, this is even more pronounced (see [Subfigure 5.8c](#)). All conditionals in Brütal Legend's vertex shaders, and 99% of Costume Quest's can be moved to the CPU. However, conditional statements are less common in shaders than typical CPU code, with the vertex shaders for Bastion, Broken Age, and Counter-Strike: Source containing none, which results in the high variability in [Figure 5.7](#).

Moving branch conditions from shaders to the CPU might allow fully specialized pipelines to be selected, thus avoiding GPU branching and removing sections of code.

5.7 Oracle Study on Constant Input Data

If a shader's inputs were known to be constant at run-time, the amount of specializable code would increase. Replacing these values with compile-time constants would allow them to be removed from their respective interfaces in the same way dead data can be removed, and would provide knock-on benefits from increasing the proportions of constant-foldable and movable code. This oracle study aims to quantify these effects by measuring the upper limits of how specializable code becomes if 100% of each input type was known to be constant.

Figures 5.9 and 5.10 show the increase in all specializable instructions, and in specializable condition variables in branch and select instructions. The following sections go into more detail about the effects of constant uniforms, inputs, and textures.

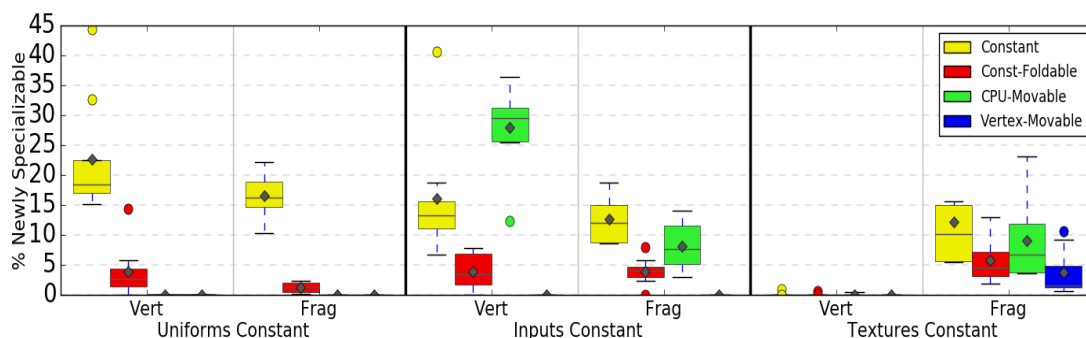


Figure 5.9: Percentage of values that become specializable if all uniforms/inputs/textures were constants. This causes fewer loads, and increases constant-foldable code. Much of the vertex shader becomes CPU-movable if vertex-buffer inputs were constant.

5.7.1 Constant Uniforms

Figure 5.9's left hand column, shows that uniform load instructions make up an average of 23 / 17% of vertex/fragment shaders, so setting these as constants greatly reduces loads. Subfigure 5.11a shows that replacing uniform loads with constants vertex shaders in simpler 2D games such as A Bird Story and Broken Age could replace up to 45% of their code. Fragment shaders experience less variability, with all games'

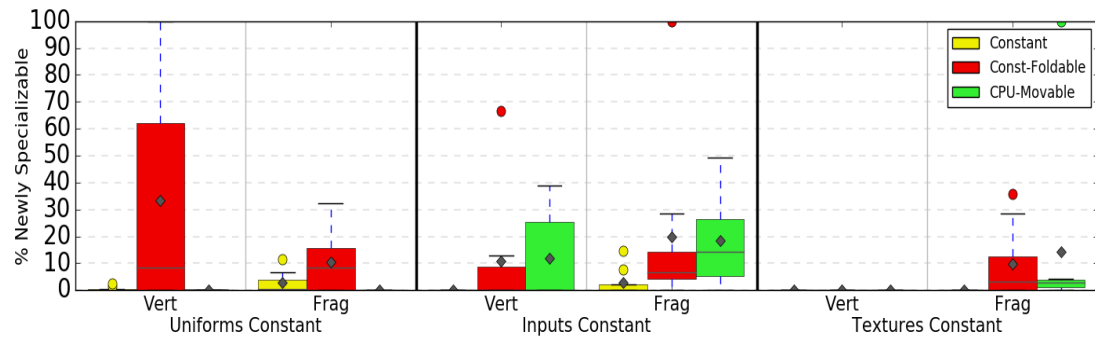


Figure 5.10: Percentage of branch/select conditions that become specializable if all uniforms/inputs/textures were constant. Many branches become constant-foldable or CPU-movable, especially when uniforms are constant.

fragment shaders being reducible by $\sim 15 - 22\%$ if uniforms were constant (see [Subfigure 5.11b](#)). Vertex shaders in more complex 3D games have a similar proportion of code dedicated to loading uniforms as fragment shaders.

There is also a small increase in constant-foldable code – 4 / 1% for average vertex/fragment shaders, and up to 14% in *A Bird Story*’s vertex shaders (see [Subfigure 5.11a](#)). This proportion is relatively small in fragment shaders, with more complex games allowing generally for slightly more constant foldable-code.

Branches

Despite comprising a small proportion of the overall instruction count, newly constant-foldable code has a large impact on branch and select instructions, as can be seen in the left hand column of [Figure 5.10](#). Here, an average of 33 / 10% of conditions can now be statically determined in vertex/fragment shaders. [Subfigure 5.11c](#) shows that for the vertex shaders in the two Buddha engine games, *Brütal Legend* and *Costume Quest*, 100% of the branches can be determined at compile time, with a further 50% of the branches in *Antichamber*’s vertex shaders. This mirrors the proportions of CPU-movable code in [Subfigure 5.8c](#). A significant proportion of code in fragment shader branches also becomes constant-foldable if uniforms are constant, especially in the two Buddha engine games. This has large code reduction possibilities depending on which branch is statically selected. [Subsection 5.8.1](#) reveals that large portions of uniforms are constant at run-time, so values close to these oracle results are likely achievable.

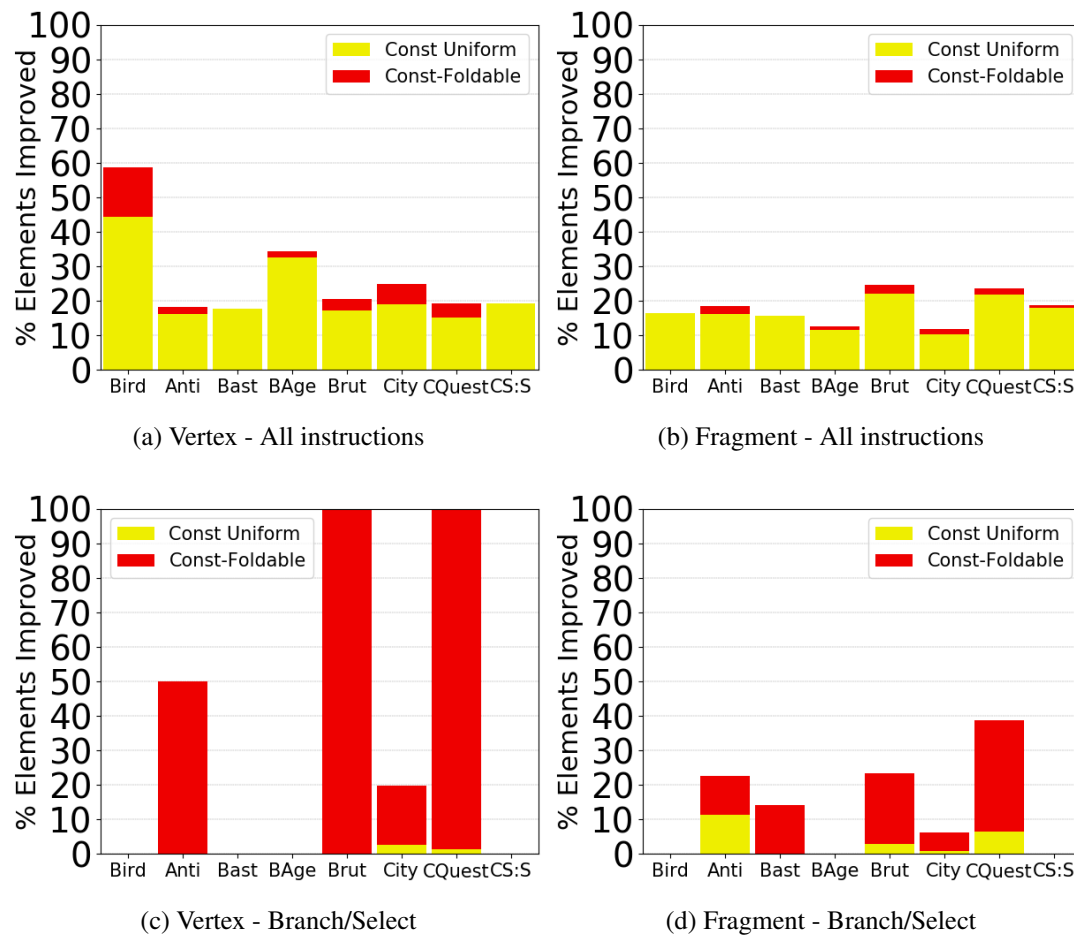


Figure 5.11: Percentage of values that become specializable given constant uniforms.

5.7.2 Constant Inputs

Figure 5.12 and the middle column of Figure 5.9 show that constant inputs are rarer than constant uniforms, but enable a wider range of specializations.

Vertex

If vertex shader inputs are constant, elements can be removed from the vertex buffer to reduce its memory footprint by an amount proportional to the number of triangles in the mesh (several thousand for complex models). On average, Vertex shaders' code is 16% input load instructions, with up to 40% in Bastion shown in Subfigure 5.12a. These loads could all be specialized away if they were constant, and would render a further $\sim 4\%$ constant-foldable. Much of the vertex shader could also be extracted to the CPU, with 28% becoming CPU-movable on average. Constant vertex buffer inputs are likely to be rare, as they require all versions of a value to be the same for

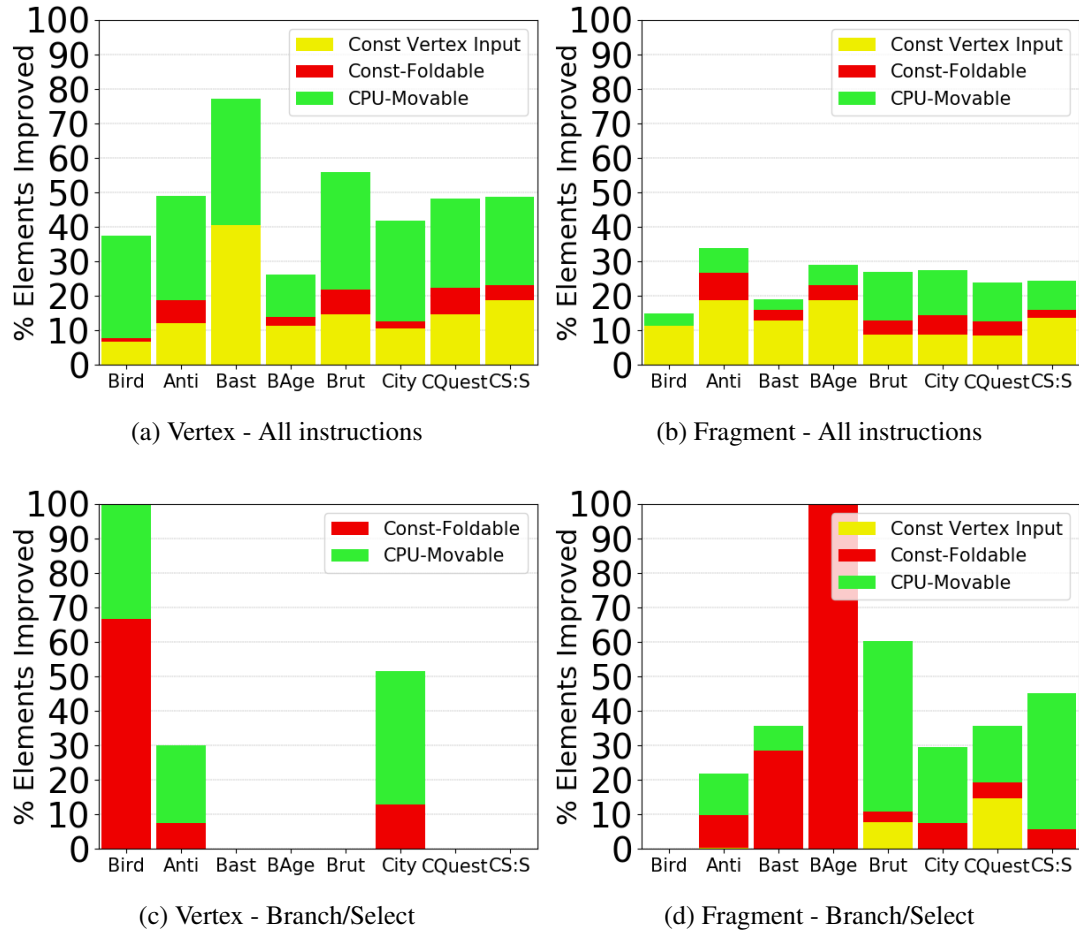


Figure 5.12: Percentage of values that become specializable given constant inputs.

all triangles in a mesh. However, in cases where they do occur, they enable many optimizations.

Fragment

Fragment shader inputs are only constant when data passed out of the vertex shader is constant, so this occurrence is also rarer than constant uniform data. If this data was constant, however, fragment shaders would benefit from removing the 13% of instruction that were input loads, increasing constant-foldability by 4%, and moving 8% of code to the CPU on average. Removing constant inputs from the vertex-fragment interface would also reduce the amount of linear interpolation required between shader stages. [Subfigure 5.12b](#) shows these values do not vary widely between games, so link-time optimizations exploiting constant-valued vertex shader outputs to optimize fragment shaders are likely to give performance benefits in many applications.

Branches

As can be seen in [Subfigure 5.12c](#) and [Subfigure 5.12d](#), constant inputs can also have a significant effect on the condition variables in branch and select instructions. [Subfigure 5.8c](#) shows that for several games, vertex shader conditionals are already statically CPU-movable, so there is less room for improvement using dynamic data. However, in games such as A Bird Story where this was not the case originally, many branches become either constant-foldable or CPU-movable if input data is constant. If all vertex buffer inputs were constant, over 70% of conditionals could be constant-folded or moved to the CPU (100% for most vertex shaders).

Branching in fragment shaders is also heavily impacted by constant input data. An average of 23% more conditions become constant or constant-foldable, reaching 100% of conditionals within Broken Age. An average of 18% of fragment shader conditionals become CPU-movable too, with up to 49% in Brütal Legend. This means around 50% of fragment shader conditions can be specialized, which is a lower percentage than that of vertex shaders, but may account for more overall, as branches are more common in fragment shaders.

5.7.3 Constant Textures

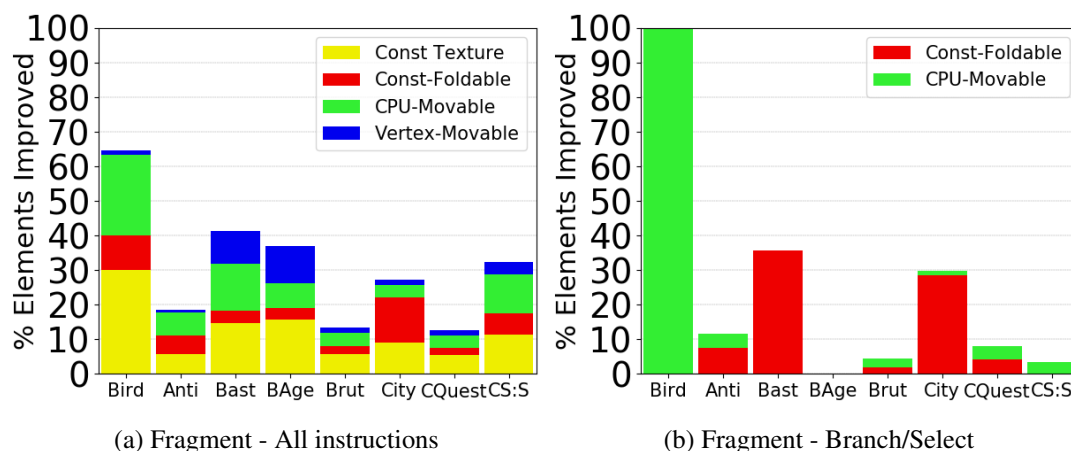


Figure 5.13: Percentage of values that become specializable given constant textures. Textures are only used in a few vertex shaders in Cities: Skylines, so are omitted here.

Consider the final column of [Figure 5.9](#). Here, texture look-ups account for 12% of fragment shader code on average, which can be omitted if their values are constant. Fragment shaders from simple 2D games such as A Bird Story consist of 30%

texture look-ups, as shown in [Subfigure 5.13a](#). Fully constant textures are rare, but constant colour channels are not uncommon, and can increase code specializability in every way. With constant textures, code becomes an average of 6% more constant-foldable, 9% more CPU-movable, and 4% more movable from fragment to vertex shaders. This variety of different specialization techniques unlocked by constant textures have a greater proportional impact on simpler 2D games such as *Broken Age*, *Bastion*, and *A Bird Story*.

Branches

Constant textures also improve branch specializability as can be seen in [Subfigure 5.13b](#). 10% more conditions become constant-foldable on average, with up to 36% in *Bastion*. In *A Bird Story*, every condition variable could be calculated on the CPU, but this increase is only 14% when averaged across all the games. This has the least impact on specializing conditionals compared to constant uniform and input data, but still offers significant potential for branch reduction.

Texture look-ups are seldom used in vertex shaders. *Cities: Skylines* is the only game tested that did so, but even then they were so rare that they had a $\leq 1\%$ impact on specializability. All significant specialization opportunities that constant textures provide are contained within fragment shaders.

5.7.4 Oracle Study Summary

Using dynamically constant data can improve code specializability far beyond what is possible with only static data. Constant input variables provide the most increase in specialization opportunities, but are the least likely to occur in practice. Constant textures are more common, and can increase specializable code of each category examined. However, they only provide benefits to fragment shaders, as textures are rarely used in vertex shaders. Utilising constant uniform data provides the largest reduction in load instructions, and can cause large percentages of branch conditions to become constant-foldable. This is true for both vertex and fragment shaders, and it will be shown in [Section 5.8](#) that constant uniforms are very common in practice.

5.8 Trace Analysis Results

Based on the previous section’s observations, this section explores to what extent uniforms are actually constant in practice, and examines the proportion of redundant API calls to update them.

5.8.1 Constant Uniform Values

Figure 5.14 shows that an average of 60-90% of uniform variables in each shader pipeline are constant for almost every game tested. This means there is room in most games to achieve close to the oracle results for perfect specializability when setting all uniforms to constant.

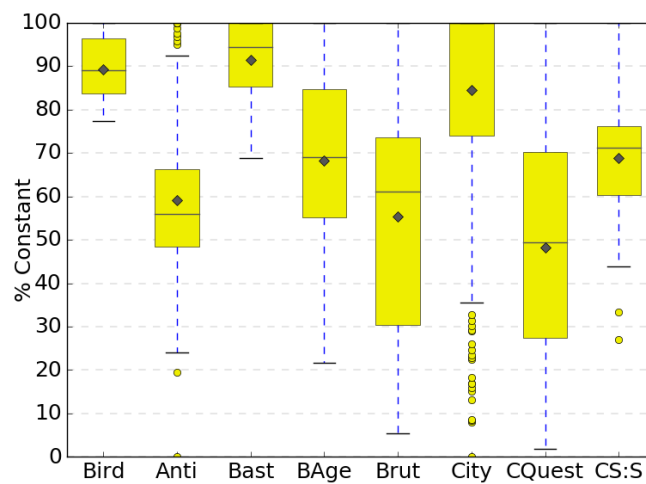


Figure 5.14: % uniforms constant at run-time for each game’s shader pipelines. All games have high proportions of constant uniforms despite their variability

In Figure 5.15, most shaders within each game have at least 50% of their uniforms constant, so these optimization opportunities are almost ubiquitous among shaders. This means that real games can achieve large reductions in uniform loading, and large increases in constant-foldable conditional statements, which allow many branches to be removed at compile-time if the dynamically constant uniform values are known.

Tools harnessing this constant data for optimizations are explored in Chapter 6.

5.8.2 Redundant Uniform Updates

In Figure 5.14 showed that large percentages of a game’s uniform variables are constant at run time, and in Figure 5.16 it can be seen that $\sim 70 - 90\%$ of all uniform updates are

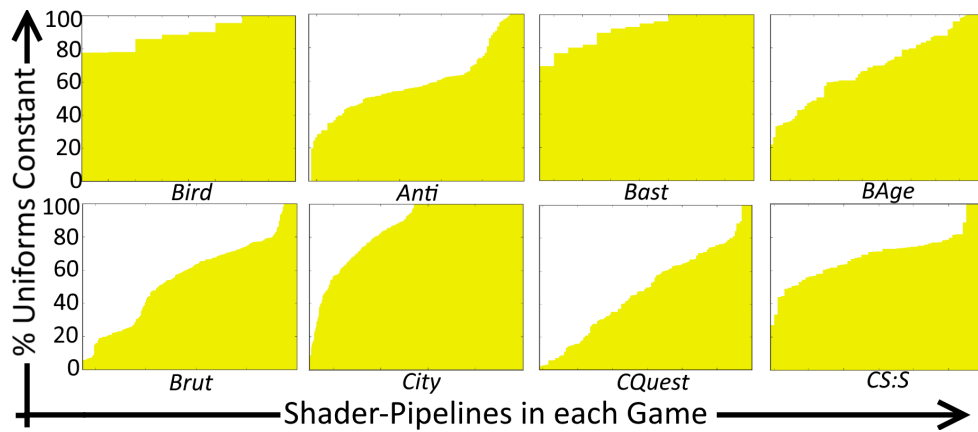


Figure 5.15: % uniforms constant at run-time for each game's shader pipelines. Games with fewer shaders such as A Bird Story and Bastion have many constant uniforms for all pipelines. The majority of every game's shaders have $> 50\%$ constant data.

redundant. This means the CPU-side code calling the update functions in the OpenGL API is not making use of the constant nature of these values, and is unnecessarily updating them with the same values many times.

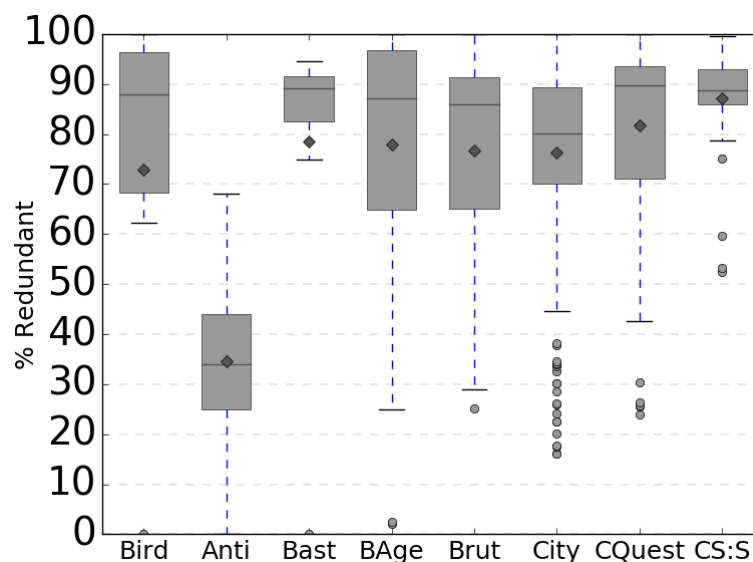


Figure 5.16: % uniform element updates redundant due to setting the same value as was previously used. For most games, most uniform updates are unnecessary.

For the majority of shader pipelines in each game (except Antichamber), over 70% of uniform update calls are redundant, as seen in [Figure 5.17](#). This means there are a large number of redundant OpenGL API calls, which might have a significant performance impact in languages such as Java on Android devices, which incurs overhead

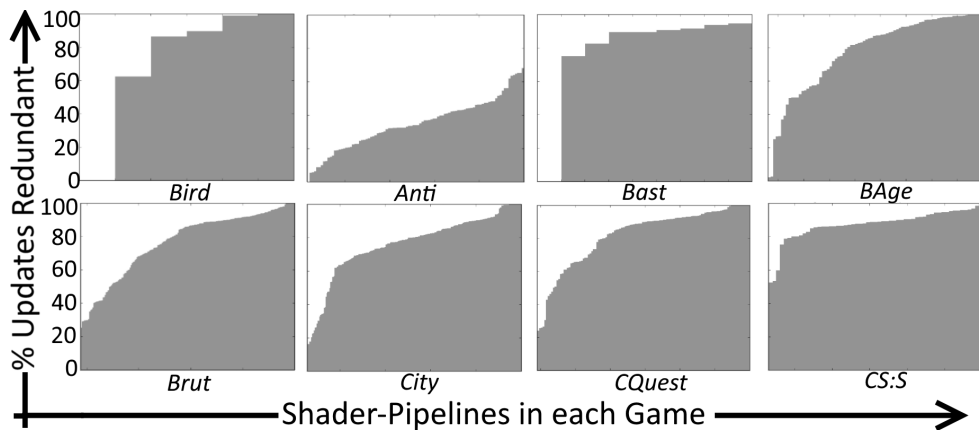


Figure 5.17: % uniform element updates redundant for all shader pipelines. In all games except Antichamber, > 70% of uniform updates for most pipelines are redundant.

for every JNI call. In the worst case, the driver may not catch this redundancy, and therefore also cause unnecessary CPU-GPU communication.

Uniforms are updated very frequently, sometimes many times per frame, so there is room for optimization by reducing these frequent redundant update calls. It may be possible to fully specialize shaders such that all constant values are removed from the uniform interface, and the remaining dynamically updated uniforms are packed more tightly together to reduce communication bandwidth and the number of API calls required to update them. It may even be possible to use this sort of dynamic data to automatically group frequently-updated uniforms together so that seldom-updated values can be cached more effectively in a separate block.

5.9 Timing Tests

In order to determine whether some of the specialization opportunities detected would be able to provide concrete performance improvements, a timing tool was built to measure speed-ups on shaders before and after transformations. This timing harness was designed to run individual shader pipelines in an isolated environment to more accurately measure their performance characteristics without them getting lost in the noise of having a full rendering trace running.

In [Figure 5.5](#), it can be seen that around 97% of Antichamber's fragment shader uniforms are unused, as well as a high percentage of its vertex shader uniforms. Upon examining these shaders' source-code, it was discovered that they all declare large arrays (224-256 elements) of `vec4` uniforms, but only tend to use the first 8 or so

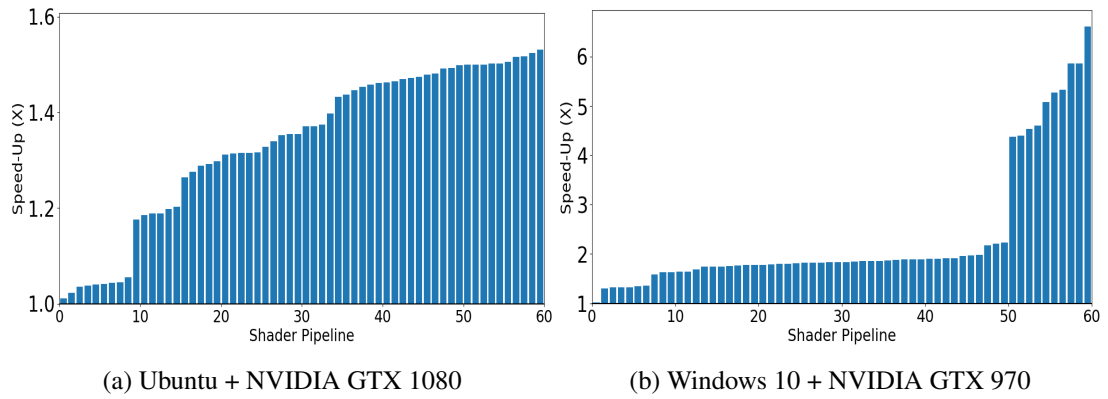


Figure 5.18: Speed-up (X) in mean CPU time per microbenchmark frame when removing dead uniforms from 60 Antichamber shader pipelines. On a GTX 970 in Windows, speed-ups were $\leq 6.6x$, with most $\sim 2x$. A GTX 1080 on Ubuntu, usually gave 1.2-1.5x.

elements. This means large proportions of the dead uniform data can be removed by simply re-declaring these arrays using a smaller size (determined by the max array index used in the source code). Some vertex shaders contain an extra uniform array for bone transformations used in skeletal animations, which is indexed into via a variable rather than a constant, so it is not possible to statically determine which elements are dead, as this would require runtime information on the array indices used in order to shrink these. However, typical uniform arrays are only ever accessed using constant indices, so it is simple to determine what the maximum index used is, and truncate the arrays accordingly.

The first 60 vertex/fragment shader pairs were extracted from a trace of Antichamber, and resized the uniform arrays to remove all trailing dead elements beyond the maximum index used. This typically meant that 2 arrays per shader pipeline with around 256 elements were shrunk to around 10 elements each.

The performance impact of shrinking these arrays was measured in a custom timing harness. This ran a simple rendering loop iterating over 1024 models in a 32×32 grid, selecting the shader pipeline, vertex attribute bindings, textures, and uniform data to use for each model, and then rendering them to the screen. Both the CPU and GPU time were recorded every frame over 100K frames, and the mean of the median 80% of times was taken to avoid outliers but still capture different phase-change behaviors.

This timing harness is similar in spirit to the isolated timing harness described in [Section 4.5](#) for fragment shaders, but its scope is expanded to include vertex-fragment pairs of shaders. Unlike the prior timing tool which drew full-screen triangles with

a fragment shader, the newer timing harness repeatedly rendered a static 3D model many times on screen to exercise both the vertex and fragment shader stages. As the model's position on screen was determined via uniform data in the form of a transformation matrix, the timing microbenchmarks needed to be customized for different vertex/fragment pairs. This ensured that the position matrix was set correctly for every vertex/fragment pair, so that the same rendering conditions were maintained for each timing run. All other uniform values were filled with default values as described in [Section 4.5](#). Instead of timing the GPU per-draw call execution time, the CPU-side time was measured per frame to capture any effects of CPU-GPU data transfers that were reduced by removing the dead data from the shaders.

The speed-ups resulting from removing the dead array elements can be seen in [Figure 5.18](#), which shows significant performance improvements on two different PCs with NVIDIA graphics cards. On a Windows 10 PC with a GTX 970, most shaders ran 2x faster, with several around 6x. The speed-ups on an Ubuntu machine with a GTX 1080 were more modest, ranging from 1.2-1.5x.

These microbenchmark results indicate that for some rendering scenarios, pruning large portions of dead uniform data can have significant performance improvements.

5.10 Conclusion

Throughout this chapter, it has been shown that specializations may be applied to shaders using static analysis alone, but greater portions of shader code may be specialized if dynamic analysis is used to detect values that are constant at run-time. Using real-world commercial games with a variety of complexities, game engines, and art-styles, it has been shown that many games have uniform data that is unused (dead), and that in some scenarios, removing this data leads to noticeable speed-ups. Inputs from vertex buffers or prior pipeline stages also contain dead data suitable for link-time optimization, especially if considering individual vector elements.

Through an oracle study, it was shown that detecting constant inputs from textures, uniforms, the vertex buffer, or prior pipeline stages can allow for many different optimizations, as well as removing many data loading instructions. Such optimizations included constant-folding, and moving code from the GPU to CPU, or from the fragment to the vertex shader (to reduce the rate at which the calculations were performed). Significant portions of branching instructions were affected by these constant-value optimizations, especially if they were uniforms.

Significant portions of shader data in the traces of each game analysed was constant at run-time, meaning that values close to the oracle predictions would be possible in practice. These constant uniform values also lead to significant redundancy in API calls being used to update uniforms, highlighting another area of potential improvement. In [Chapter 6](#), the promising opportunities detected here about constant uniform values are capitalized on, and measured in the context of full execution traces.

Chapter 6

Optimizations Within Full Execution Traces

6.1 Introduction

This chapter explores optimizations at the scope of a full execution trace, rather than on individual shader pipelines. Using the findings from [Chapter 5](#), trace-level optimizations were implemented to take advantage the large proportion of constant uniform data detected. The results here cover a wide range of popular games, including those that utilize uniform buffer objects (UBOs) instead of traditional uniforms (see [Subsection 2.6.3](#)). Instead of timing isolated shaders, whole frames were profiled to more accurately assess the real-world benefits of these techniques.

One initial optimization explored was to remove all redundant CPU-side uniform update calls, as these formed significant proportions of uniform updates (see [Subsection 5.8.2](#)). However, removing these calls had a near-negligible effect on performance, despite significantly boosting some games' trace playback speed (see [Subsection 6.4.1](#)).

Another more successful optimization performed was to collect all constant uniform values, and constant-fold them into the source-code of all shaders. The shader strings submitted to OpenGL were then replaced with these specialized constant-folded variants, enabling performance increases in several games (see [Subsection 6.5.2](#)).

This chapter begins with a discussion of the 17 benchmark games selected based on their popularity, technical complexity, and variety of game engines and art styles. [Section 6.2](#) also covers the process used to extract traces from them. [Section 6.3](#) describes the trace analysis and modification tools developed. It explores the technical

challenges involved in tracking uniform data throughout a trace, exporting it for constant folding, and then matching the modified shader to the correct source-specifying API call. [Section 6.4](#) explains how to overcome the pitfalls involved in extracting timing data from replaying traces. These timing techniques are then used to record the performance measurements reported in [Section 6.5](#), which also covers the percentage of constant data detected for each new benchmark game.

6.2 Benchmark Games

In this chapter, the number and variety of benchmark games is expanded to ensure that the initial results from [Chapter 5](#) generalize across various real-world execution examples. This section aims to introduce this new wider selection of benchmarks, which are used for all performance measurements in subsequent sections. [Subsection 6.2.1](#) explains how these games were selected, and [Subsection 6.2.2](#) describes the technical process of capturing execution traces from these games for analysis.

6.2.1 Selecting Games

Steam is the world’s largest PC game retailer, with around 75% of the global market-share[447]. Of the 233 Linux-compatible Steam games available to me, 17 were selected as benchmarks based on both popularity and technical interest.

Popularity

To determine each game’s popularity, SteamDB[448], an independent game analytics site, was used. This site provided statistics such as their estimated number of downloads over the game’s lifespan (see [Figure 6.1](#)), and the peak number of concurrent players each day (see [Figure 6.2](#)). After ranking games according to these numbers to ensure both popularity and relevancy, the final benchmarks were picked to provide a variety of engines, art styles, and genres. Compatibility with the analysis tool-chain was also an important criterion.

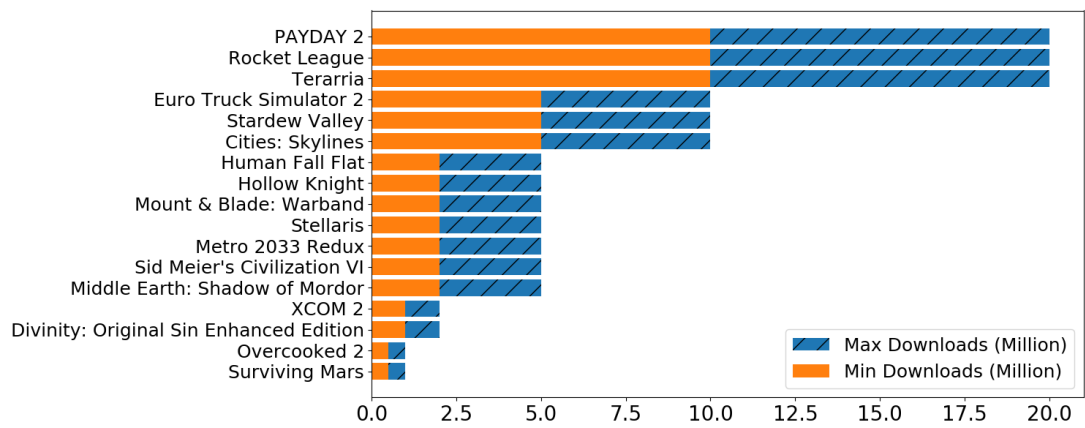


Figure 6.1: SteamDB's estimated minimum/maximum downloads (million) per game

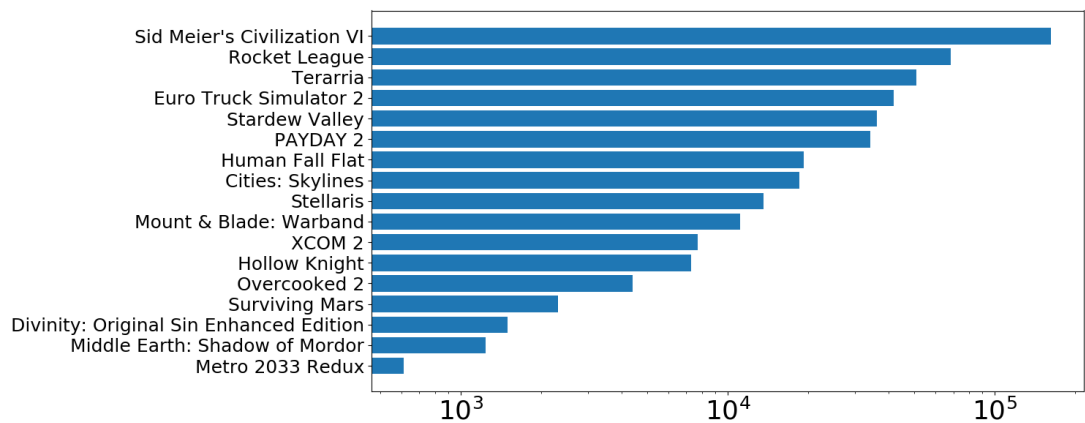


Figure 6.2: Peak concurrent players over 24 hours on Jan 08th 2020

Engine Variety

The final selection of games, and their respective engines is shown in [Table 6.1](#). Complex 3D games were favoured over simpler sprite or tile-based 2D games, which are seldom technically strenuous, so offer fewer opportunities to optimize. As such, only three popular 2D games were selected. As explained in [section 2.4.2](#), Unity accounts for over 50% of the game-engine market share, and its cross-platform capabilities make it a popular choice for Linux-compatible games. To allow for more engine variety, only 4 Unity-based games were selected, each showcasing different genres and art styles. Aside from the 2 Unreal Engine games, the rest primarily utilize custom-built in-house game engines from larger studios, which ensures technical variety.

Game Name	Engine/ Framework	Developer	2D/ 3D	UBOs?
Terarria	XNA	Re-Logic	2D	-
Middle Earth: Shadow of Mordor	LitTech	Monolith	3D	✓
Cities: Skylines	Unity	Colossal Order	3D	-
Rocket League	Unreal	Psyonix	3D	-
Sid Meier's Civilization VI	-	Firaxis	3D	-
Stardew Valley	-	ConcernedApe	2D	-
Divinity: Original Sin Enhanced Edition	Divinity Engine	Larion Studios	3D	✓
Metro 2033 Redux	4A Engine	4A Games	3D	✓
Stellaris	Clausewitz	Paradox	3D	-
Surviving Mars	-	Haemimont	3D	✓
Euro Truck Simulator 2	Prism3D	SCS Software	3D	-
Mount & Blade: Warband	-	TaleWorlds	3D	-
Hollow Knight	Unity	Team Cherry	2D	-
Overcooked 2	Unity	Ghost Town	3D	-
Human Fall Flat	Unity	No Brakes	3D	-
PAYDAY 2	Diesel 2.0	OVERKILL	3D	-
XCOM 2	Unreal	Firaxis	3D	-

Table 6.1: Games used as benchmarks, their game engines, developer name, peak concurrent players over 24 hours on Jan 08th 2020, and estimated number of owners.

Shader/Pipeline Counts

Figure 6.3 shows the number of vertex and fragment shaders used for rendering in traces from each game, as well as the number of program pipelines created by linking these shaders together. These statistics provide a rough proxy for technical and graphical complexity.

Some games have the same number of vertex and fragment shaders as linked programs. This means that each vertex/fragment shader pair is only ever linked together into a single unique program pipeline. In games where these counts differ, the same shaders are re-used between multiple pipeline programs. This may be a single fragment shader used with multiple different vertex shaders, or vice versa, as there is no

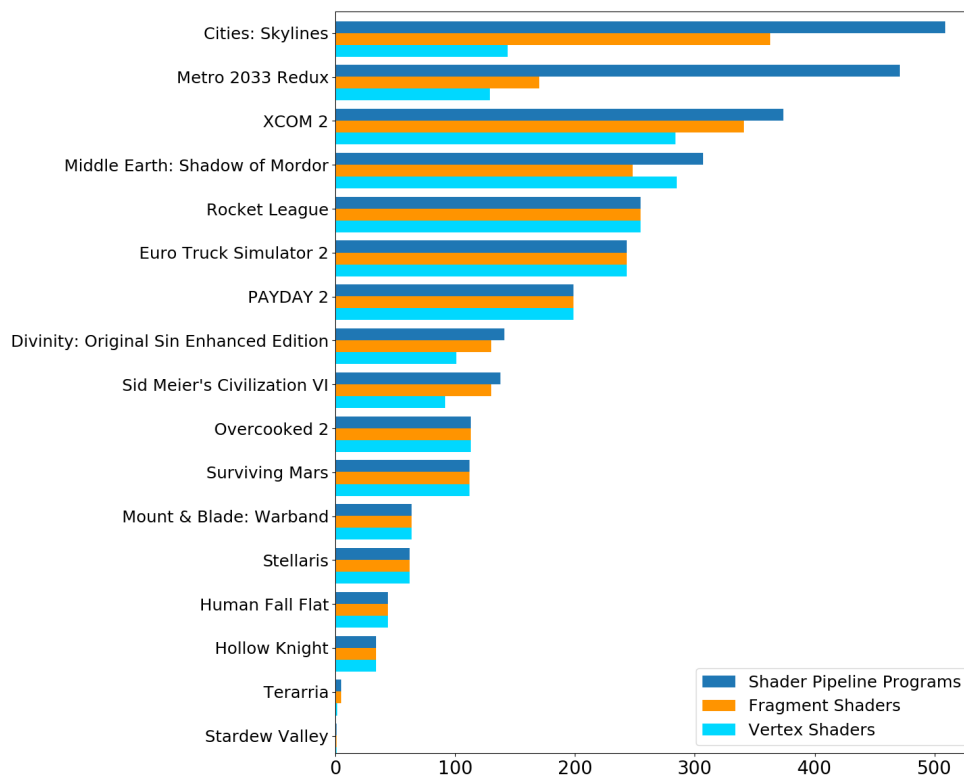


Figure 6.3: Number of unique shaders and combined shader pipelines.

limit to the number of programs each shader may be linked to.

Whether or not shaders are re-used between different pipelines has implications when it comes to the constant folding passes later on. Uniforms are associated with linked pipeline programs, rather than the individual shaders that comprise them. If a shader is shared between two pipelines with differing sets of constant uniform data, this shader will need to be split into two different specialized variants - one for each set of constant uniforms. Adding these additional specialized variants would add complexity to the implementation, and potentially some performance detriments as well.

Uniform Transfer Method

The final column in [Table 6.1](#) indicates whether or not each game uses uniform buffer objects (UBOs). UBOs are an alternative method of uploading uniform data in large batches and re-using the data across multiple pipelines. The differences between UBOs and traditional uniforms are explained more in [Section 2.6](#). As UBOs were not available within older versions of OpenGL, they are typically only used in newer, more technically demanding titles. The Unity engine has the capability to use UBOs, but the traces recorded from the selected Unity games did not utilize them.

Summary

The information in the above tables shows that the games selected are both popular and technically diverse, all having at least 0.5 million downloads and thousands of recent players. They come from a variety of developers and game engines, and are a mix between 2D and 3D games. The number of shaders and programs in each game varies drastically, as does their uniform transfer method, which indicates that the benchmarks cover a variety of technical complexities and implementation techniques. These games also cover different genres, art styles, levels of graphical realism, and camera angles, including top-down strategy games, first-person shooters, and 2D side-scrolling platformers, so cover a wide range of graphical styles too.

6.2.2 Capturing Traces

This section describes the procedure for capturing game execution traces, along with the issues encountered and workarounds developed. For each game, the aim is to capture a short gameplay trace showcasing the main mechanics and typical levels, scenes, characters, and visual effects to gather realistic data patterns. These were captured using `apitrace`[\[12\]](#), an open-source tool capable of recording OpenGL function calls into a compressed trace file that can later be replayed or analysed.

Tracing Games with `apitrace`

The `apitrace` tool is designed to capture traces from applications by injecting itself as a replacement for the OpenGL API library. On Linux, it uses `LD_PRELOAD` to override symbols for each of the system's OpenGL API functions. It provides a wrapper around OpenGL, which records each function call before passing it on to the real library to execute. For simple applications, this can be achieved using the command:

```
apitrace -o ./outputTrace ./programToTrace
```

Commercial games are often less simple to trace. Some lack a single easily-executed file, instead offering a "splash screen" which opens up first to allow users to set various options on a simple GUI before the main game opens up. The presence of splash screens can cause difficulties tracing applications, as `apitrace` may fail to latch on to the real game process if it attempts to attach itself to the splash screen's process first.

Steam's Debugger Interface

Many 64-bit games also can be traced reliably directly via Steam using its built-in interface for injecting debugger programs as a game launches. The Steam client is 32-bit, so does not interfere with apitrace's library injection for 64-bit games. Steam offers a command-line option to supply a `DEBUGGER` environment variable to initialize a debug program before running games. An apitrace command can be provided as a `DEBUGGER`, and Steam will automatically call it when launching the game with the correct library paths and Steam capabilities, making it more likely to launch without crashing.

When tracing via Steam, all Steam processes must be closed before running the launch command. This ensures that apitrace can hook into the game's process correctly. The Steam client can launch games via the command line by using their product ID, which is publicly available in the URLs of the games' store pages. For example, the command to trace Hollow Knight via the Steam client is:

```
DEBUGGER="./apitrace trace -o ./outputTrace" steam steam://  
  runtimeid/367520
```

One caveat to this process is that when the Steam client is closed, the recorded trace becomes corrupted, and crashes any tools that try to read it. To avoid this, the trace file can be copied to a different location before the client is closed. This problem may be due to serialization of multiple traces into a single output file, so another possible workaround is to omit `-o ./outputTrace` from the command.

Summary

For the 17 benchmark games selected, a combination of the above techniques was sufficient to extract traces, with Steam's `DEBUGGER` interface being the most reliable method for the majority of games. Several other popular games were initially selected as benchmark candidates, but either crashed when apitrace attempted to inject itself, or recorded traces with severely corrupted visuals. DOOM and DOTA, for example, both use advanced texture compression techniques that apitrace is unable to replay reliably, so were not included in case these defects affected performance results. However, the majority of games were able to be traced correctly either via the Steam client, or using the external environment setup scripts described above.

6.3 Tools Developed

To determine whether shader specialization using constant uniforms is able to speed up real-world software, execution traces from numerous games were recorded, analysed, modified, and timed. This section describes the analysis and modification tools developed to gather these results.

The open-source `apitrace`[12] framework was used extensively for capturing and timing traces, and was also extended to include custom transformation passes. The `LunarGlass`[10] offline shader compiler was also extended to modify shaders extracted from these traces, allowing the constant uniform data to be folded into them. The following sections describe these custom `apitrace` and `LunarGlass` passes in more detail, and various implementation details required to track whether uniform data remains constant throughout an OpenGL program.

6.3.1 Overview

[Figure 6.5](#) provides a rough outline of the different phases shaders must go through in this chapter's experiments. The initial `apitrace`-based analysis stages require keeping track of the graphics programs/pipelines created, and source-code for each shader attached to each pipeline. The details of tracking this information throughout a trace are described in [Subsection 6.3.2](#).

Once shaders and pipelines are tracked, information about their associated uniforms can also be recorded throughout the trace. The process for tracking traditional uniform data is described in [Subsection 6.3.3](#), and the more complex process of tracking data stored in uniform buffer objects (UBOs) is described in [Subsection 6.3.4](#).

After the initial analysis passes have determined which uniforms remain constant throughout the lifetime of the trace, this information is exported and passed on to `LunarGlass`-based shader compilation tools to specialize the shaders. [Subsection 6.3.5](#) explains how uniform data recorded from traces can be matched to uniforms within shader source-code, and then used for extensive constant-folding within the custom `LunarGlass` pass. It also describes several additional optimizations made possible when constant uniform values are known.

Once these specialized shaders have been generated using the information about constant uniforms, they can then be re-inserted into a the trace file, and the execution speed can be timed to determine how beneficial this specialization process can be. This timing and measurement phase of the process is covered in [Section 6.4](#).

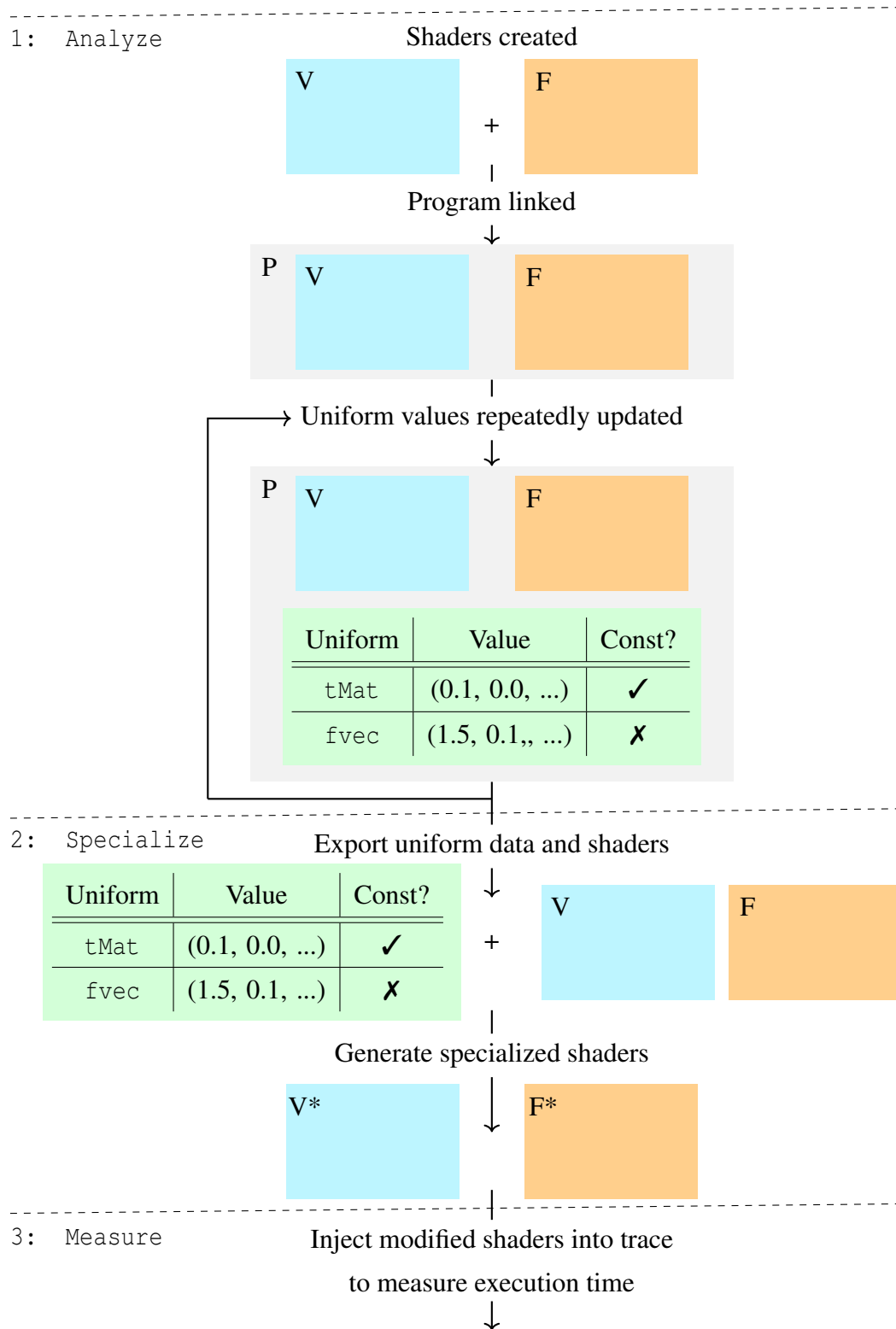


Figure 6.5: The shader life-cycle is tracked during trace analysis to detect constant uniforms. Specialized shaders are then generated, re-injected into the trace, and timed.

6.3.2 Tracking Shaders and Programs

Tracking OpenGL shader and pipeline state forms the backbone of many of this chapter's apitrace analysis and modification passes. This requires recording not just the current OpenGL state, but its history as well. Doing so allows constant uniforms to be detected, and source-strings to be associated with shaders, even after deletion.

Program Objects

One critical piece of OpenGL state is the current "program", and the shaders which comprise it. OpenGL follows a single-threaded design philosophy, so a single active current program is assigned as global state, which then determines which shaders are to be used in subsequent draw calls.

In OpenGL terminology, a program is an object that can have several shader objects attached to it. Once the attached shaders are compiled, the program can be linked together, and represents a single pipeline. Although some games eschew programs in favour of separable shader pipelines, the majority still use monolithic program objects. To simplify the analysis passes, only games with linked programs are considered.

Reusable Object IDs

Most OpenGL objects have a single integer identifier, which represents a unique object in the current OpenGL state. However, these identifiers do not remain unique throughout the entire trace lifetime. When a shader is deleted, its ID is marked as available for reuse, so any subsequently created shaders may be assigned the same ID.

In a real-time OpenGL driver, only the current state is important, so this ID reuse is not a problem. However, for the analysis passes required in this chapter, each shader must be uniquely associated with a single shader string. As such, it is necessary to track a separate globally unique identifier for each shader, in addition to the OpenGL ID used to refer to it by function calls.

[Figure 6.6](#) demonstrates two fragment shaders reusing the same OpenGL ID after one is deleted. Separate records are stored with different globally unique IDs, despite the shared OpenGL ID. Each record contains information such as the shader's source string, the number of draw calls it was used with, and whether it has been compiled. A new record struct is generated for every `glCreateShader` call irrespective of the shader's OpenGL ID, ensuring identifier reuse never causes data to be overwritten.

<pre>f0 = createShader(FRAG) shaderSource(f0, "strA") compileShader(f0) deleteShader(f0) f0 = createShader(FRAG) shaderSource(f0, "strB") compileShader(f0)</pre>	<pre>ID = Frag 0 openGL_ID = 0 compiled = ✓ next = NULL prev = NULL num_draws = 0 src = "strA"</pre>	<pre>ID = Frag 1 openGL_ID = 0 compiled = ✓ next = NULL prev = NULL num_draws = 0 src = "strB"</pre>
---	--	--

Figure 6.6: Two independently created fragment shaders happen to re-use the same OpenGL ID after one is deleted. Separate structs are generated when `createShader` is called, allowing both versions to be recorded despite the shared OpenGL ID.

Shader Object Lifecycles

This chapter's analysis passes require shaders to be associated with a single string of source code. This approach differs from OpenGL's conception of shader objects, where multiple source strings may be assigned throughout a shader's lifespan. The same shader object may be compiled, attached, and linked into a program, and then detached, recompiled with new code, and reused in an entirely different program object. In some games, a shader's source string may also be overwritten with a small placeholder value after compilation to free up memory. [Figure 6.7](#) illustrates some of the possible permutations of events that shaders may undergo, which can complicate tracking.

These scenarios require careful tracking to ensure the correct source string is associated with each shader record. Tracking the API call that specified each shader source string is necessary to allow shaders to be replaced with modified variants for the experiments in [Section 6.5](#).

Tracking Shader Source Strings

To account for the fact that OpenGL shader objects may be given multiple source strings throughout their lifespan, separate records are generated every time a source string is specified, even when assigned to the same shader object (see [Figure 6.8](#)).

New shader records are created under two scenarios. Firstly, as in [Figure 6.6](#), a new record is initialized when `glCreateShader` is called. Initially, the source string value is NULL, and it can be initialized with a call to `glShaderSource`.

The second trigger for initializing a new shader record, is when `glShaderSource` is called for a shader with a non-NULL source string (as in [Figure 6.8](#)). This new

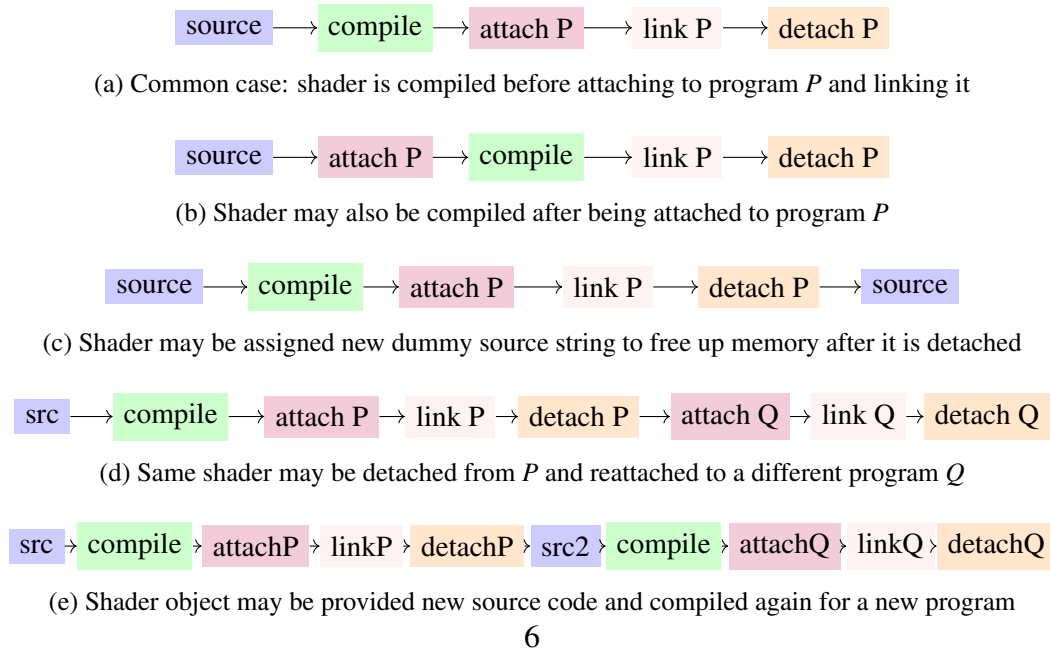


Figure 6.7: Example shader lifecycles tracked throughout traces. Many combinations of assigning source strings, compiling, attaching, linking, and detaching must be handled.

record copies the previous shader's OpenGL ID, but tracks separate values for its source string, number of draws, and compilation status.

Each record represents a snapshot of the same shader object at a different point in time, rather than independent shader objects that happen to reuse the same OpenGL ID as in Figure 6.6. To avoid losing this information, a doubly linked list of shader records is created with each entry pointing to the `next` and `previous` snapshots in history (see Figure 6.8).

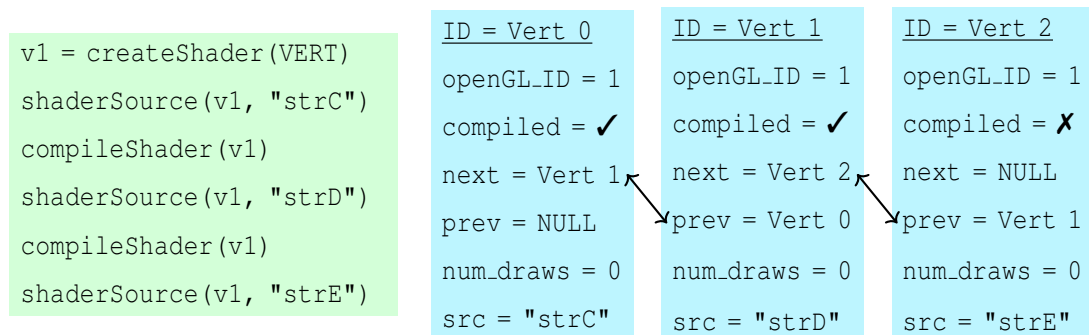


Figure 6.8: A single OpenGL vertex shader object is repeatedly assigned new source strings and recompiled. A linked list of three separate structs records the history of every shader source string in case it is ever linked into a program. A new record is generated when `shaderSource` is called for shaders with pre-existing source strings.

Tracking Program Objects

Program objects are more straightforward to track, as although their OpenGL identifiers may be reused, the program objects themselves tend not to be redefined after they are linked. The games examined here only ever link together program objects once, so no historical data about previous linkage states is tracked. As such, it is sufficient to simply generate a new record with a globally unique ID whenever `glCreateProgram` is called, similar to the fragment shader example in [Figure 6.6](#). This handles the case where the same OpenGL ID refers to different program objects after one is deleted, and the ID number is re-used.

Attaching and Linking Shaders to a Program

The most important state recorded for a program is the list of linked shaders which comprise it, and the uniform values associated with it. Although programs in the games examined are only ever linked once, shaders may be attached and detached freely, both before and after linkage. As such, it is necessary to track not only the shaders linked in the program, but also those currently attached to it. An example of the state recorded for a program object is shown in [Figure 6.9](#).

<pre>p0=createProgram() attachShader(p0,f0) attachShader(p0,v1) linkProgram(p0) useProgram(p0) drawArrays(...) drawArrays(...)</pre>	<u>ID = Prog0</u> openGL_ID = 0	
	uniforms = ... ubos = ...	
	num_draws = 2	
	<u>Attached Shaders</u>	
	<u>ID=Vert2</u>	<u>ID=Frag1</u>
	openGL_ID=1	openGL_ID=0
	compiled= X	compiled= ✓
	next=NULL	next=NULL
	prev=Vert1	prev=NULL
	num_draws=0	num_draws=2
	src="strE"	src="strB"
<u>Linked Shaders</u>		
<u>ID=Vert1</u>	<u>ID=Frag1</u>	
openGL_ID=1	openGL_ID=0	
compiled= ✓	compiled= ✓	
next=Vert2	next=NULL	
prev=Vert0	prev=NULL	
num_draws=2	num_draws=2	
src="strD"	src="strB"	

Figure 6.9: A program attaching and linking the shaders from [Figure 6.6](#) and [Figure 6.8](#). Note that although `Vert2`, the current version of `v1` was attached, the most recent compiled version `Vert1` is used when the program is linked. Attached and linked shaders are separate lists, and only linked shaders matter during draw calls or uniform updates.

Linked shaders and attached shaders must be stored in two separate lists. Linked

shaders are the only important values for all future analysis passes, and are fixed whenever `glLinkProgram` is called. The list of currently attached shaders is transient, and is only retained to ensure the correct shaders are recorded when the program is linked.

To determine which shaders are linked, it is insufficient to simply copy from the attached shaders list into the linked shaders list. The shader records in the attached list all represent snapshots of the shader state at a specific point in time. As such, it is necessary to traverse their history pointers to find the most recently compiled version.

First, the chain of `next` pointers is followed until the most recent version is selected. Then, the list is traversed backwards via the `prev` pointers until a compiled shader is detected. This allows shader source strings to be tracked, even if the same object is overwritten and recompiled multiple times. This compiled version of the shader is placed permanently in the program's list of linked shaders, and remains unaltered, even if the shader is detached and recompiled again.

State-Tracking Example

A demonstration of this shader history traversal is shown in [Figure 6.9](#). The vertex shader `v1` was generated using the sequence of instructions in [Figure 6.8](#). As such, its shader source at the time it was attached to program `p0` was the uncompiled string `"strE"` (stored in the shader record `Vert2`). However, when `glLinkProgram` is called, the linked list of historical shader states is traversed (shown in [Figure 6.8](#)), and the shader record `Vert1` is determined to be the most recent compiled version of vertex shader `v1`. As a result, `Vert1` is placed in the list of `p0`'s linked shaders, despite `Vert2` being the version that was initially attached.

All per-program state such as uniform and UBO information is associated only with source strings from linked shaders. Likewise, when the two draw calls are issued in [Figure 6.9](#), only the `num_draws` counters for the program and linked shaders are updated. Despite the attached shader `Vert2` being a future version of `Vert1`, only `Vert1`'s counter is updated, as these historical snapshots represent different shader source strings that may be used independently. Many shaders may be compiled and never used, so this draw call tracking mechanism allows for all unused shaders to be excluded from analyses such as the graph in [Figure 6.3](#).

6.3.3 Tracking Uniform Data

The purpose of all the shader and program state recording in [Subsection 6.3.2](#) is to be able to track uniform data assigned to a program, and then associate it with a shader source string. Uniforms are always associated with a single program object, and their values are retained even when the program is not currently active.

Uniform Locations

Uniforms can be assigned to variables within shaders using unique integer “location” identifiers determined by the compiler, as explained in [Subsection 2.6.2](#). This enables optimizations such as eliding locations of uniforms removed by dead code elimination, or packing data by interleaving uniforms from different shaders from the same pipeline.

Developers must use `glGetUniformLocation(progID, name)` to query uniform variable locations before assigning them. Valid uniforms return a positive integer, and invalid uniforms, such as those elided by the compiler, will return `-1`. OpenGL traces record all these location queries, and this chapter’s analysis passes utilize them to infer the existence of uniforms and determine their names.

Mapping Uniform Locations to Names

All OpenGL functions which assign uniform value must refer to each uniform using its location. However, in order to specialize the shader’s source code, the name of each uniform must also be tracked (see [Subsection 6.3.5](#)). Textual name strings provide a unique identifier for variables that is cross-compatible between the apitrace analyses, and the LunarGlass optimization passes. Tracking the name string value of the `name` argument from calls to `glGetUniformLocation` can generate a map between uniform names and locations, as shown in [Figure 6.10](#).

Tracking Aggregate Uniforms

Aggregate uniform types such as arrays and matrices are common within OpenGL, especially those containing multiple `vec4` elements. Before UBOs were created, `vec4` arrays were commonly used to update large blocks of uniform data in a single call, and this technique remains common in many engines. As seen in [Figure 6.10](#), arrays and matrices may span multiple locations, typically one per `vec4`-sized element.

OpenGL’s specification does not guarantee that consecutive array elements will be assigned consecutive locations. As such, conformant applications must query the

```

0 = glGetUniformLocation(p0, "tMat")
2 = glGetUniformLocation(p0, "tMat[2]")
4 = glGetUniformLocation(p0, "fvec")
5 = glGetUniformLocation(p0, "tex")

```

Prog0 Uniforms:		
Name	Loc	Idx
tMat	0	0
	2	2
fvec	4	0
tex	5	0

Figure 6.10: Mappings between uniform names and locations generated by tracking `glGetUniformLocation` calls. Note that arrays or matrices may span multiple locations.

location of every individual array index before assigning values to them. In these cases, simple string parsing of the queried uniform's name is sufficient to build a map between sub-indices of named uniforms and their locations.

Unfortunately, not all games conform to this aspect of the OpenGL specification. Some non-conformant games assume that array indices for `vec4` arrays are consecutive, and simply query the first index and then extrapolate all subsequent locations. In these cases, heuristics during value assignment are required to associate previously unseen uniform locations with the correct variable and index information.

Tracking Constant Uniform Values

To track which uniforms remain constant throughout a trace, their values must be recorded, along with the number of times they are updated. Uniform updates use typed functions such as `glUniform4fv`, which specifies a pointer to (potentially multiple) floating-point `vec4` elements to upload to the specified uniform location. For every uniform, a list of all currently assigned values is stored. This list has an associated type and stride based on the function used to update it (e.g. `glUniform4fv` for floating-point `vec4`s, or `glUniform2iv` for signed integer `ivec2`s). Using these strides and the list of per-sub-index uniform locations, a single array of scalar values can be maintained for every uniform, as shown in [Figure 6.11](#).

To determine whether a uniform remains constant, a set of update counts is tracked alongside its current value. Whenever the uniform is updated, its current and prior values are compared, and each element's update count is increased whenever different values are assigned. After the entire trace has been analysed, all elements that remained constant will have an update count of either 0 or 1.

Prog0 Uniforms:			
Name	Stride	Values	Updates
tMat	4	1, 1, 1, 1,	1 1 1 1
		0, 0, 0, 0,	0 0 0 0
		0, 0, 0, 0,	0 0 0 0
		4, 0, 0, 4	1 1 1 1
fvec	3	0.5 0.8 0.9	1 2 2
tex	1	8	1

Figure 6.11: Uniform values and their update counts tracked via `glUniform...` calls. Note the differing per-element update counts for `fvec` in location **4**, and how location **3** is inferred to be an element of `tMat` despite not being queried in [Figure 6.10](#).

Uniform Tracking Example

[Figure 6.11](#) provides an example of uniform update counts being tracked. The uniform `fvec` in location **4** is assigned two different `vec3` values, but the first element of both is 0.5, so this only counts as being updated once. This allows the first element of `fvec` to be treated as a constant value of 0.5, and used in fine-grained constant folding, even though the other vector elements change throughout the trace.

For fully OpenGL-conformant games, [Figure 6.10](#)'s location map is sufficient to assign values to the named uniforms in [Figure 6.11](#). However, some games assume that `vec4` array elements are assigned consecutive uniform locations, and only query the array's base location. In [Figure 6.11](#), location **3** is assigned a value, but was never queried by `glGetUniformLocation`. To account for this case, updates to previously unseen locations are assumed to be assigning values to sub-indices of the nearest available uniform. In this case, the closest uniform whose base location is below **3** is `tMat`. Using `tMat`'s base location of **0**, the stride of 4 inferred from the `glUniform4fv` call, and assuming consecutive locations for array indices, it is possible to resize `tMat`'s list of values. Location **3** is mapped to sub-index 3 of `tMat`, and values are copied in. All intermediary indices in the value and update-count arrays are filled with zeroes.

Using these techniques, values and update counts for every scalar element of every uniform variable can be tracked throughout the lifetime of the trace. Both the final uniform values and their update counts can then be exported to LunarGlass to allow it to perform constant-folding on a per-element basis for uniforms with the matching name (see [Subsection 6.3.5](#)).

6.3.4 Tracking UBOs

Unlike the traditional per-program uniforms described above, Uniform Buffer Objects can contain arbitrary binary data, and may be read by multiple different programs (see [Subsection 2.6.3](#) for details). Determining constant values within UBOs for each program requires tracking two different sets of data - the contents of all buffers, and the values the program reads from bound buffers during draw calls.

Identifying Buffers

OpenGL buffers use a similar numerical ID scheme to programs and shaders. Only the values read from these buffers can impact specialization, so the buffer contents can be treated as transient state. Thus, using OpenGL buffer identifiers is sufficient, and globally unique IDs or historical snapshots of buffer contents are not required.

OpenGL functions identify buffers using two methods. Firstly, named buffers may be identified using their OpenGL integer ID. The second method refers to a series of global "slots" that buffers may be bound to. Each slot has an enum identifier for the type of array bound to it, such as `GL_ARRAY_BUFFER` for per-vertex attributes or `GL_UNIFORM_BUFFER` for UBOs. Although it is possible for the same buffer to be bound to multiple different types of slot, the benchmarks examined generally did not reuse UBOs with different binding types. Both name and slot-based buffer identification methods may be used interchangeably.

Tracking Buffer Contents

Only the buffers' current state must be tracked, which requires handling several scenarios:

- Creating buffers via `glCreateBuffers` or `glGenBuffers`
- Binding buffers to the correct buffer slots in global state via `glBindBuffer`
- Resetting buffer contents via `glBufferStorage` or `glBufferData`
- Rewriting subsets of the buffers via `glBufferSubData`
- Copying between buffers via `glCopyBufferSubData`
- Clearing buffer contents via `glClearBuffer*`

- Memory mapping buffers via `glMapBuffer` or `glMapBufferRange`
- Updating memory-mapped regions via fake `memcpy` calls in the trace

Once the correct buffer has been selected, its binary contents can be tracked by intercepting all the above buffer update scenarios, and copying, creating, or deleting various sections of a transient copy of the buffer contents in memory.

Binding Buffer Sub-Ranges

In addition to tracking buffers' binary contents, it is also necessary to track which ranges of the buffer have been bound. To specify which UBO contents to upload to the GPU, a game must bind sections of UBOs to subindices of the global `GL_UNIFORM_BUFFER` binding slot. When a draw call is issued, the bound sub-ranges of the buffer's contents can be copied in to a separate array representing all the values a shader has seen for that particular UBO. The UBOs in a shader's interface are extracted via a simple text parser examining the shader source strings, and the `GL_UNIFORM_BUFFER` subindices corresponding to each UBO can be tracked by intercepting calls to `glGetUniformBlockIndex` and `glUniformBlockBinding` which may update bindings, or query bindings implicitly set by the shader compiler.

Tracking Per-Program UBO Values

Once the correct subrange of buffer contents has been matched to a corresponding UBO in a shader's interface, a similar analysis to the traditional uniforms in [Subsection 6.3.3](#) can be applied. By comparing each byte of the UBO's current contents to the newly bound values at the time of the draw call, it is possible to track how often each byte of UBO data has been updated to a different value. If the byte has been updated 0 or 1 times by the end of the trace, it can be regarded as dynamically constant data. The results of this analysis are presented in [Subsection 6.5.1](#).

6.3.5 Creating Specialized Shaders

Once the values and update frequencies of all per-program uniforms and UBOs are tracked, they can then be written to file for subsequent constant-folding operation in LunarGlass. This section explains how the constant data is folded in, as well as some additional optimizations implemented to take advantage of common cases.

6.3.5.1 Matching Uniforms

To fold the correct uniform data into the shader source code, a mechanism is required to match the flat arrays of uniform values tracked through the traces to the list of LLVM loads in LunarGlass. LunarGlass represents reading uniform data as load instructions from global variables in a specific address space. Information such as the uniform's name and type is tracked via LLVM metadata. Complex types such as nested structures would require full traversal of this type tree to associate the correct elements together during constant folding. However, the benchmarks examined here only needed scalars, vectors, and arrays, which simplified the implementation.

As explained in [Subsection 6.3.3](#), uniform updates to different locations within arrays are matched to single base-location. This allows all data for a single uniform array to be associated with a single variable name, and enables consistent array index numbering to access it. The LunarGlass constant-folding pass can match the array variable's name from the LLVM metadata to the corresponding name exported from trace, and specify the correct sub-element of the array data using the index provided by the LLVM `getElementPointer` instruction. Using variable name strings and flat array indexing avoids relying on the compiler implementation-defined concept of uniform locations, allowing unambiguous variable access during constant-folding.

6.3.5.2 Replacing With Constants

Once the correct uniform name and array index are selected, the update frequency of each loaded element can be checked. If 0 or 1 updates occurred during the course of the trace, the value is assumed to be constant. For scalars, the entire value can be trivially replaced with a constant. The only exceptions to this are integers representing texture samplers, which cannot be assigned constant literals within shader source code. To avoid this case, it is necessary to examine LunarGlass's type metadata, and avoid constant-folding uniforms marked with texture sampling flags. All other constant scalar uniforms can be fully replaced.

The majority of loaded uniforms are vectors. These are often only partially constant, and contain at least one variable element. Opting to only specialize fully constant vectors could avoid potential performance losses caused by additional move instructions required to specialize loaded vectors. However, this option would miss out on the largest portion of constant data, as well as subsequent optimizations possible for special cases such as 0s or 1s. To maximize the chance of potential speed-ups, the chosen

implementation specializes both fully and partially constant vectors in the hope that the GPU driver's compiler will reduce performance losses by using inexpensive vector swizzle operations and loading less unnecessary data.

In LunarGlass's LLVM IR, partially constant vectors can be handled by loading in the full vector, extracting any non-constant elements, and inserting them into another variable constructed from the constant-foldable data. After performing dead code elimination, and translating back into GLSL, the resulting vectors are generally constructed from swizzled components of the loaded vector.

An example of this per-component specialization is shown below:

```
vec4 result = uniformVar * otherVar;
```

If the trace shows every component of `uniformVar` except `y` is 0.0, this becomes:

```
vec4 temp = vec4(0.0, uniformVar.y, 0.0, 0.0);
vec4 result = temp * otherVar;
```

6.3.5.3 Additional Optimizations

The values 0 and 1 occur frequently within constant uniforms, which enables additional optimizations to occur on partially constant data. One common example of this is transformation matrices which only represent scaling and translation, rather than rotation, so only contain non-zero data in their main diagonal and rightmost column. Uniforms used as boolean toggles, or floating-point weighting factors also commonly contain the values 0 or 1 when they are used to enable or disable optional effects.

LunarGlass can only constant-fold fully constant vectors, so additional optimizations were implemented to take advantage of partially constant data. These cover recurring patterns of addition and multiplications with mostly zero vectors.

Multiplication

One optimizable scenario is the multiplication of vectors where all but one of the elements is zero. In this case, the single non-zero element can be multiplied as a scalar instead, and used to construct the result vector using constant zeros for the other elements instead if necessary:

This optimization can be applied to the constant-folding in the previous example:

```
vec4 temp = vec4(0.0, uniformVar.y, 0.0, 0.0);
vec4 result = temp * otherVar;
```

$$(0\ b\ 0\ 0) \times (x\ y\ z\ w) \longrightarrow (0\ y \times b\ 0\ 0)$$

Figure 6.12: Compacting multiplication for vector elements with zeros

This can be transformed to use only scalar multiplication and vector construction:

```
vec4 result = vec4(0.0, uniformVar.y * otherVar, 0.0, 0.0);
```

Depending on the implementation of the GPU driver's compiler, this may transform the loads and multiplications of 2 `vec4`s into the load and multiplication of 2 scalar variables, potentially saving cache space, registers, and instructions cycles.

Addition

Another common case involves vector addition. When two vectors are added together, such that each non-zero element in one vector corresponds to a zero element in the other, the addition can be replaced with a vector construction as shown below:

$$(0\ y\ 0\ 0) + (x\ 0\ z\ w) \longrightarrow (x\ y\ z\ w)$$

Figure 6.13: Compacting addition for vector elements with zeros

This would transform the following code:

```
vec4 vecA = vec4(0.0, a.y, 0.0, 0.0);
vec4 vecB = vec4(b.x, 0.0, b.z, b.w);
vec4 result = vecA + vecB;
```

into the simplified form of a single vector construction using swizzled elements:

```
vec4 result = vec4(b.x, a.y, b.z, b.w);
```

6.4 Timing Techniques

This section describes how the execution times of different traces were measured and compared, and how various measurement problems were overcome. In previous chapters, individual shaders and pipelines were timed in isolation. This chapter, however, covers mass changes to many shaders at once, so the performance impact is measured across entire frames from real execution traces.

The traces used here only record OpenGL API calls used for rendering, so do not fully capture all aspects of a game's performance. However, rendering is typically

the most computationally intensive section of any game's per-frame execution time, and additional CPU-side calculations generally run in parallel to the main OpenGL rendering thread. As such, measuring speed-ups on the game's rendering code provides a good proxy to the expected performance impact on the game as a whole.

Originally, the timing experiments in this chapter were intended to be measured on all frames of a full execution trace. However, due to several drawbacks described in [Subsection 6.4.1](#), repeatedly replaying a smaller number of individual representative frames was chosen for timing traces instead. This approach, and how it avoids the problems with full trace replays, is described in [Subsection 6.4.2](#).

All timings in subsequent sections were measured on a 64-bit Linux machine running Ubuntu 20.04.1 LTS, which had an NVIDIA GTX 970 GPU, an Intel i7 6700K CPU, 16GB of 2666MHz DDR4 RAM, and a SanDisk Ultra 3D 500GB SSD with read speeds of up to 560MB/s.

6.4.1 Full-trace timings

The apitrace framework contains several built-in profiling capabilities. However, many of these capabilities are intended for pinpointing hot-spots within the trace, rather than producing stable timing results suitable for comparisons between executions. This section describes several potential measuring methods available within apitrace, the problems these techniques can introduce, and why the repeated frame timing method described in [Subsection 6.4.2](#) was chosen to avoid these problems.

Per-Draw-Call GPU Timings

One timing method that apitrace allows is to submit GPU timer queries around every draw call, which can give a very fine-grained performance analysis of individual shader pipelines. However, this adds significant profiling overhead, which may eclipse any measured speed-ups. It may also cause the GPU driver to schedule draw-calls in a more sequential manner to cope with the large number of GPU timer queries, thus making the test less representative of typical execution.

Per-Frame CPU Timings

To avoid these overheads, per-frame CPU-side timer queries were used instead of the per-draw-call GPU-side timers to better measure the overall improvement per-frame.

This significantly reduced overheads, as only 2 timer queries were inserted for potentially thousands of other function calls in each frame. Measuring CPU-side time also avoids altering any scheduling of GPU-side operations by the driver, as no additional OpenGL instructions for timer queries are introduced. It also aims to capture the performance of the entire rendering system and its interaction with the OpenGL API, and not just the GPU's shader execution performance.

Problems with Full Trace Timings

Using apitrace's built-in per-frame CPU profiler to time entire traces also introduces problems due to file IO and decompression overheads. As seen in [Figure 6.14](#), the trace files examined were typically between 1 and 4 GB in size. They are compressed using snappy[449], a byte-oriented LZ77-type[450] compression algorithm from Google. Snappy is designed for rapid compression and decompression speeds, rather than optimal output file-sizes. It is fast enough to enable apitrace to record traces at interactive frame rates, and play them back even faster. However, despite being optimized for speed, loading in multi-gigabyte trace files and decompressing them on the fly introduces significant overheads. For many OpenGL functions, the CPU time spent loading, decompressing, and parsing the function call outweighs the time it takes to execute the function.

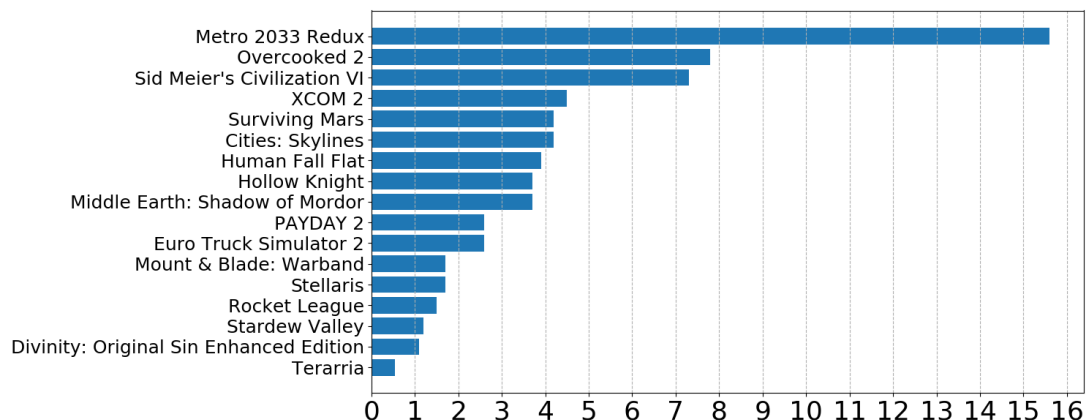


Figure 6.14: Size (GB) for each trace recorded from the benchmark games.

False-Positives when Removing Redundant Functions

The file I/O and decompression overheads described above both masked legitimate performance gains, and incorrectly amplified the impact of other optimizations. False

positive results were generated using full-trace per-frame CPU timings to measure the impact of optimizations that removed function calls.

Subsection 5.8.2 determined that in many games, the majority of uniform update calls are redundant. This also holds true for the games in Section 6.2, such as Euro Truck Simulator 2, and Mount & Blade: Warband, where redundant calls comprised 76.72% and 86.55% of all uniform updates respectively. Many games also contained several redundant calls to `glUseProgram` every frame, which could be elided.

Removing the redundant uniform updates and `glUseProgram` calls from these traces provided unexpected speed-ups of 43% and 44% respectively over typical game-play frames (after all loading and menu screens had occurred). However, these results were incorrect. Removing these redundant function calls significantly reduced the traces' file-sizes, hinting that the performance gains were from reduced I/O and decompression overheads rather than from improving the OpenGL rendering speed.

To investigate these spurious speed-ups, another built-in feature of apitrace was used - repeatedly rendering a trace's final frame. Unlike during typical trace replays, the calls from this final frame are decompressed and parsed once, then stored in memory for all subsequent uses, rather than being repeatedly loaded from disk. This eliminates I/O and decompression overheads at the cost of only measuring times for a single frame. The fact the same frame is repeatedly rendered also reduces the effects of noisy data, as a stable average can be produced if it is replayed sufficiently often.

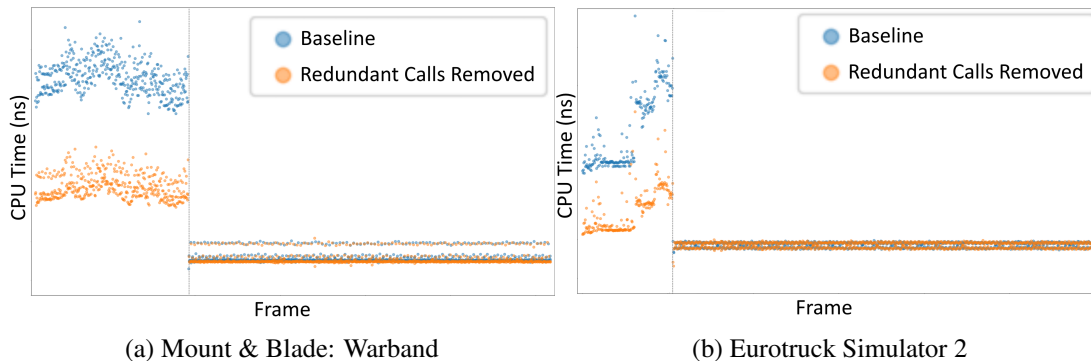


Figure 6.15: Per-frame CPU timings for traces before and after unnecessary uniform updates and `glUseProgram` calls were removed. Both games' initial frames show $\sim 44\%$ speed-ups while loading and decompressing calls on the fly. However, when repeating the final frame from within memory 1000 times, this difference vanishes.

The timings in Figure 6.15 show that the large performance gains initially visible from removing redundant calls will completely disappear when replaying the final

frame from within memory. These timings were measured after truncating each trace to ensure their final frames contained gameplay, rather than the original final frame's graphics tear-down code. In the initial portion of each graph, the $\sim 44\%$ performance improvement from removing redundant uniform updates is clear. However, when repeating the final frame from memory 1000 times, this performance difference disappears, indicating that the impact of removing the redundant OpenGL calls is negligible.

The difference between the initial frames, and the final repeated frame is $\sim 1.5x-3x$, which demonstrates the significant overhead that I/O and on-the-fly decompression and parsing has during benchmarking. The initial $\sim 44\%$ speed-ups were caused purely by optimizing trace playback by reducing the number of functions to load and decompress, and had minimal impact to the game's underlying rendering performance.

Legitimate Speed-Ups Overshadowed by Overheads

The I/O and decompression overheads in trace playback not only introduced false-positive results, but also overshadowed genuine performance improvements. [Figure 6.16](#) depicts the effect of constant-folding known uniform data into traces from Hollow Knight. Timing the trace's first 20,000 frames gives a noisy $\sim 3.6\%$ speed-up, whereas repeating the game's 10,000th frame for 10,000 iterations shows a more stable and visible performance improvement of 5.5%. The gap between full-trace and repeated-frame timings in Hollow Knight is less pronounced than for Euro Truck Simulator 2 or Mount & Blade: Warband, but the overhead is still visibly reduced. This is likely due to Hollow Knight being a simpler 2D game, and thus requiring fewer OpenGL calls to be loaded from disk each frame.

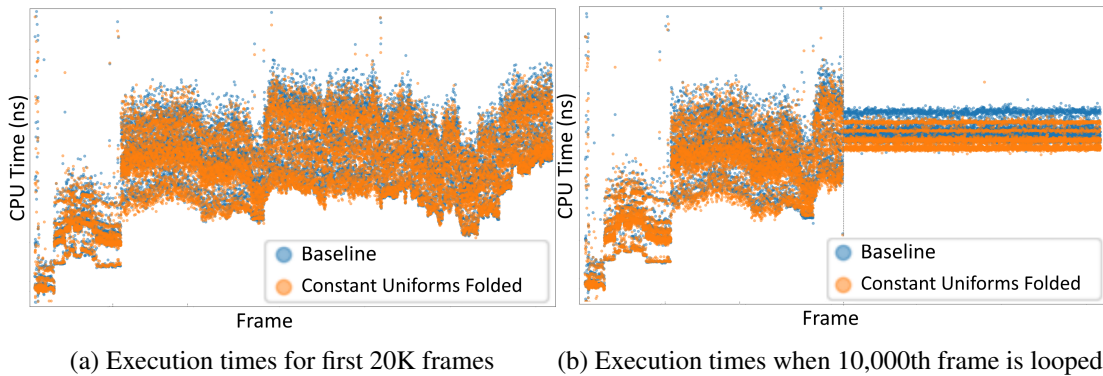


Figure 6.16: Hollow Knight frame times after constant-folding uniforms. Repeating a frame produces more stable, lower overhead results with more discernible deltas.

These findings indicate that timing full traces with apitrace leads to unreliable results. To mitigate the large artificial overheads and unrealistic performance characteristics that occur when timing an entire trace as it is read from disk, timings gathered from single repeated frames are used exclusively as the primary method for timing the performance impacts of our trace optimizations.

6.4.2 Repeated Frame Timings

Repeatedly timing a single frame eliminates unnecessary I/O and decompression overheads, which avoids false-positive performance improvements, and makes real speed-ups more visible. It also allows more stable results to be extracted from noisy data if the frame is repeated many times. However, it limits the scope of the measurement to a single frame of a trace, rather than the entire game-play sequence, so several representative frames need to be selected to ensure sufficient coverage. This section describes how traces can be trimmed to select specific final frames, and how different distributions of timing results can be compared between frames to measure the magnitude of improvements.

Trimming Traces to Selected Frames

To ensure representative frames were benchmarked, six snapshots of game-play were manually selected for each game to showcase a wide range of common rendering scenarios. Apitrace only allows a trace's final frame to be repeatedly rendered, so traces were trimmed to end at the final call of the target frame. To truncate traces, apitrace provides an interface for trimming up to specific function call ID. It also provides tools for exporting screenshots of every frame, along with the call ID terminating that frame, which can be passed in to the trimming tool once a target frame is chosen.

Usually, a game's first few thousand frames are loading screens or menus where different save files are selected. They may also have opening cutscenes that are typically pre-rendered videos rather than dynamically rendered 3D models, so are not representative of typical in-game real-time rendering workloads. As such, many initial frames can be ignored. However, all initial frames must be included in the final trimmed traces to ensure OpenGL is in the correct state for the final repeated frame several thousand frames later. Once generated, trimmed traces can be replayed using apitrace's benchmark setting, which disables optional error-checking calls that might impact performance. The final frame is looped ten thousand times to generate stable

results.

Comparing Timing Results

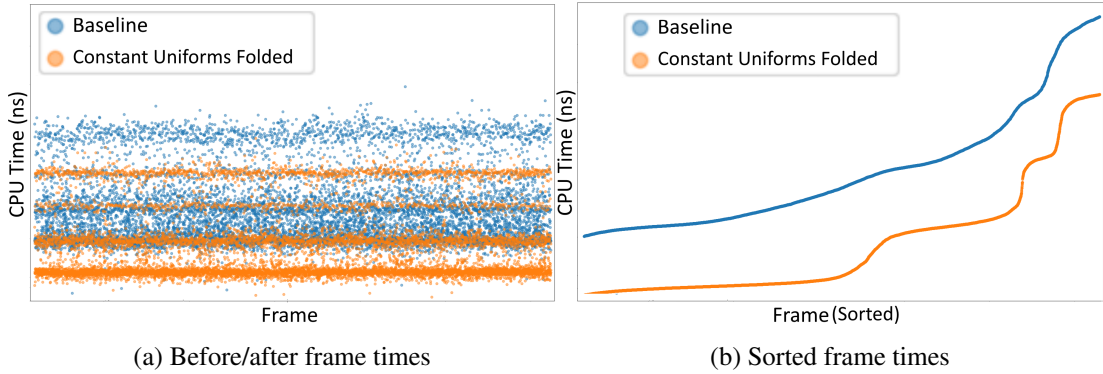


Figure 6.17: Disparate scatterings of frame timings are sorted before comparison to quantify the effects of optimizations as the normalized percentage delta between them.

Comparing trace timings is often more complicated than simply estimating a median time for each trace and comparing them directly. [Subfigure 6.17a](#) demonstrates that although repeated frame timings are relatively stable, they often fall into multiple bands. Some optimizations may affect the distribution of these bands, not just their central point, so more than just their medians must be compared.

To compare these distributions, frame times are sorted as shown in [Subfigure 6.17b](#) so that the lower and higher bands in each time-series can be compared against one another. The normalized differences between each point of the two sorted time-series are calculated after pruning the upper and lower 10% to avoid outliers. These distributions of normalized percentage deltas are displayed in [Figure 6.20](#)'s violin plots, and all subsequent summary percentages represent the medians of these distributions.

6.5 Performance Results on Whole Execution Traces

This section shows the experimental results of running the benchmark games from [Section 6.2](#) through the apitrace-based analysis tools and LunarGlass-based constant-folding passes described in [Section 6.3](#). After re-injecting the constant-folded shader code back into the benchmark traces, timing techniques from [Section 6.4](#) are used to profile the altered traces and measure the performance impact of the optimizations.

6.5.1 Constant Data

Using the tools described in [Section 6.3](#), data for both traditional uniforms, and within uniform buffers could be tracked, along with the number of updates performed that altered each element's value. From this, the proportion of uniform data that remains constant throughout the entire trace's execution can be calculated. This technique does not prove that it is safe to fold these constants into shaders, as all potential game-play scenarios are not exhaustively enumerated. However, it provides a rough indication of what proportion of data remains unchanged throughout typical game-play, and would therefore potentially benefit from shader specialization.

[Table 6.2](#) shows that the traditional uniforms for all benchmark games were at least 35.2% constant, with many having 60% or higher. Only shaders for programs that were used for at least one draw call were included within this calculation. Other shaders that were compiled but never used were omitted to avoid artificially inflating these numbers. These results reinforce the findings of [Section 5.8](#) by extending them to a wider range of additional popular games in a variety of different game engines, and showing similar trends in the proportions of constant data they use. The distributions of constant uniform data across all shader pipelines in each game shown in [Figure 6.18](#) also follow similar patterns to [Section 5.8](#), with most pipelines having at least 40% constant uniforms.

[Table 6.3](#) and [Figure 6.19](#) show that the 4 games using UBOs show similarly high levels of constant data to those using traditional uniforms, with broadly similar distributions. For most of these games using UBOs, the traditional uniform data used is over 95% constant. This is because these games use UBOs as their primary delivery mechanism for uniform data, and only use traditional uniforms for texture sampler slot indices (see [Subsection 2.6.5](#)), as these cannot be stored within UBOs. Texture slot indices typically remain consistent throughout most of the programs, so this high proportion of constant data is unsurprising. *Divinity: Original Sin Enhanced Edition* is the exception to this principle, as it mixes both modes of delivering its uniform data, so has a lower overall percentage of constant data for its traditional uniforms, as they are not reserved solely for binding texture samplers.

Overall, these high percentages of constant uniform data indicate the potential for many shaders to be specialized within all the games examined.

Game Name	Uniform Elements	Constant Elements	% Const
Terarria	145	122	84.14
Middle Earth: Shadow of Mordor	1927	1845	95.74
Cities: Skylines	50084	22914	45.75
Rocket League	145782	80763	55.4
Sid Meier' Civilization VI	50823	24569	48.34
Stardew Valley	21	14	66.67
Divinity: Original Sin Enhanced Edition	8101	5404	67.47
Metro 2033 Redux	777	774	99.61
Stellaris	6510	4183	64.25
Surviving Mars	493	493	100.0
Euro Truck Simulator 2	17167	7626	44.42
Mount & Blade: Warband	13447	8366	62.21
Hollow Knight	1637	1325	80.94
Overcooked 2	10545	7558	71.67
Human Fall Flat	2623	1687	64.32
PAYDAY 2	20642	7266	35.20
XCOM 2	345763	188996	54.66

Table 6.2: % of constant uniform elements in all shaders used in each game.

Game Name	UBO Bytes	Constant Bytes	% Const
Middle Earth: Shadow of Mordor	1244688	947764	76.14
Divinity: Original Sin Enhanced Edition	48000	27614	57.53
Metro 2033 Redux	441824	169155	38.29
Surviving Mars	290460	159000	54.74

Table 6.3: % of constant uniform buffer bytes in all shaders used in each game.

6.5.2 Timing Results

Due to issues with LunarGlass compatibility, timing results were only gathered for 12 of the 17 benchmark games. Also, to avoid the complexity of creating multiple specialized sub-variants of shaders depending on which programs they were linked

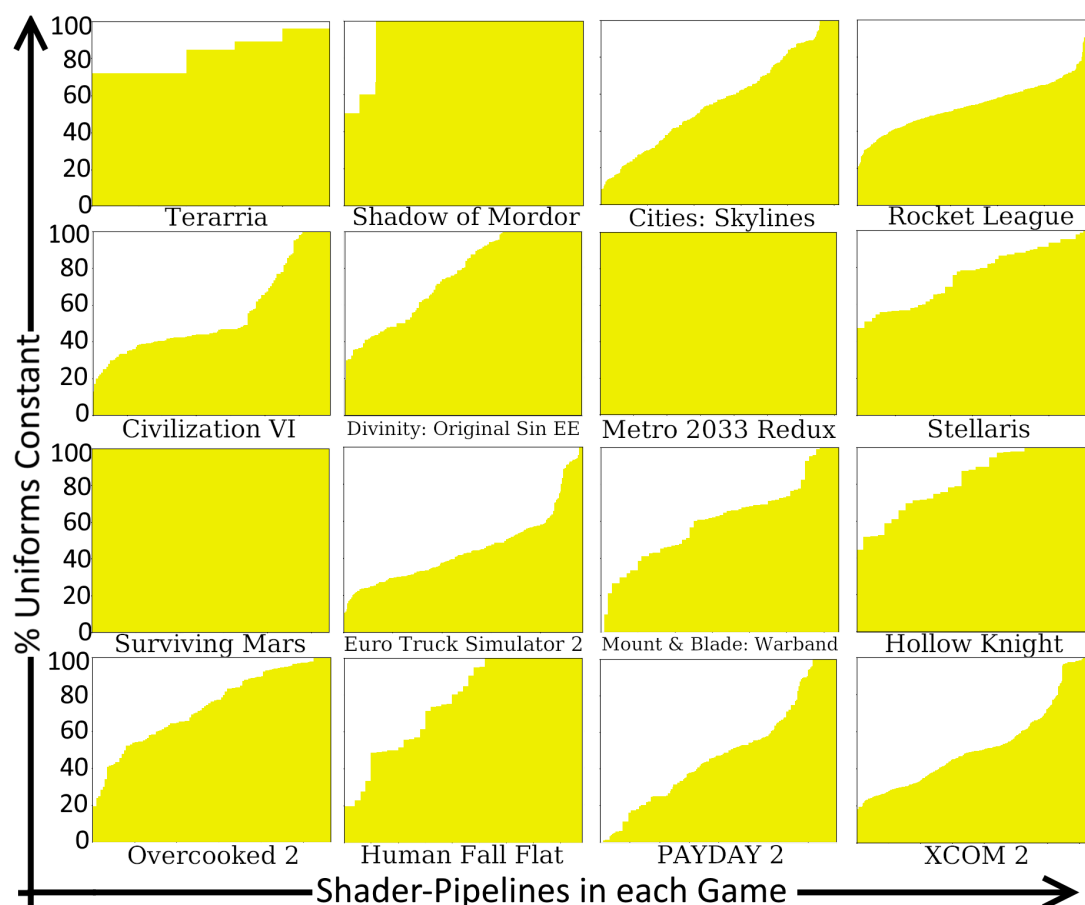


Figure 6.18: % of constant uniform elements in all shaders used in each game.

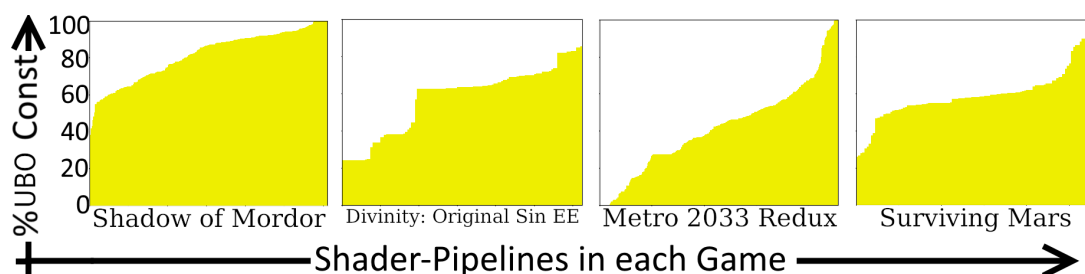


Figure 6.19: % of constant uniform buffer bytes in all shaders used in each game.

to, specialized shader variants were only generated for shaders linked into a single pipeline. In games where the number of vertex and fragment shaders matches the number of program pipelines (see [section 6.2.1](#)), all shaders can be specialized. For games where these numbers do not match, only a subset of shaders could be replaced, but this typically covered over 50% of shaders.

From this subset of 12 games, 6 frames were selected from each to represent typical game-play scenarios. These frames were repeated 10,000 times during benchmarking,

and these experiments were repeated 5 times to ensure results were stable, and thermal or OS scheduling effects could be mitigated during these repeated trials.

The distributions of percentage speed-ups for each frame of each game are presented in [Figure 6.20](#). Screenshots of the selected frames, along with the median percentage speed-up for that frame are displayed on the subsequent 4 pages to provide context to these speed-ups.

[Figure 6.20](#) shows that most games experienced at least a marginal performance improvement. Games using the popular Unity engine (see [section 2.4.2](#)) generally fared well, with one frame from *Overcooked 2* reaching an over 25% speed-up, and its other frames achieving 12% – 15% boosts. [Figure 6.26](#) shows that the frame that improved by 25% included a lightning-flash effect, and a pop-up indicating that the level had started, so it is likely one of the shaders performing these full-screen effects was significantly improved by specialization. *Hollow Knight*, another Unity game, achieves around 3% – 5% improvement, with the exact improvement varying depending on the complexity of the scenes (see [Figure 6.23](#)). *Human Fall Flat* experiences 1.5% – 3% improvements, with simpler, more zoomed-in shots of the character improving more than wider shots encompassing more of the level’s geometry (see [Figure 6.24](#)).

Euro Truck Simulator is a non-Unity game that experiences significant speed-ups, especially in detailed close-up shots of the truck’s exterior, which was boosted by almost 10% (see [Figure 6.22](#)). Other shots of more typical driving scenarios also experience $\sim 5\% - 6\%$ speed-ups. *Mount & Blade: Warband* also experiences speed-ups of $\sim 1\% - 3\%$, with shots containing many close-up characters improving more than those featuring more empty terrain (see [Figure 6.25](#)). *Rocket League*, an Unreal Engine game, gains around 1% improvements on most frames (see [Figure 6.28](#)). As many of the shaders were skipped due to LunarGlass incompatibilities or being linked to multiple programs, these results might be higher if all shaders were fully specialized. *Stellaris* also experiences some small speed-ups of $\sim 0.6\%$, but experiences slow-downs of similar magnitudes when the screen contains the mostly empty backdrop of a solar system (see [Figure 6.30](#)).

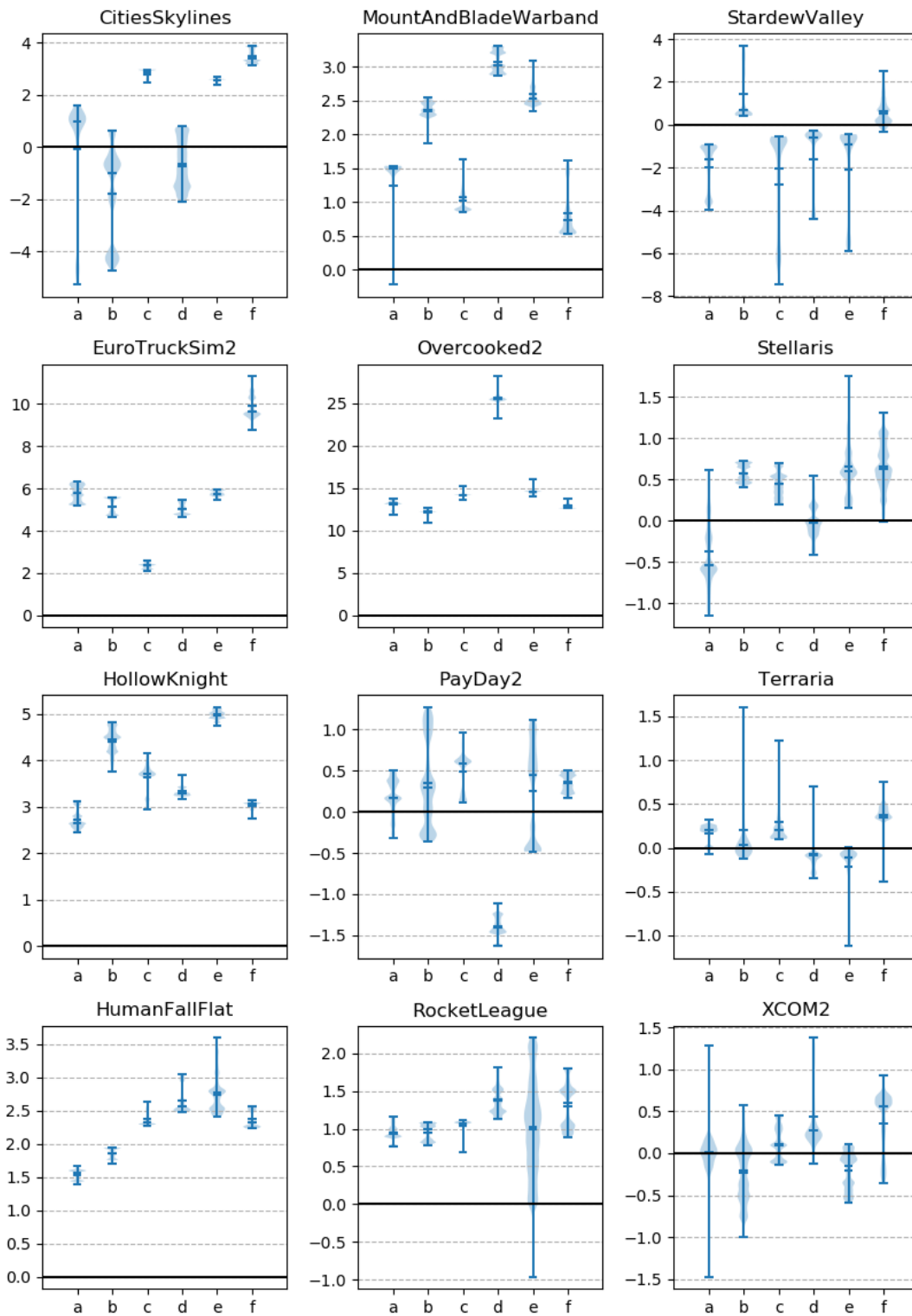


Figure 6.20: Percentage speed-up for six frames for each game (higher is better).



Figure 6.21: Cities Skylines



Figure 6.22: Euro Truck Simulator 2



Figure 6.23: Hollow Knight

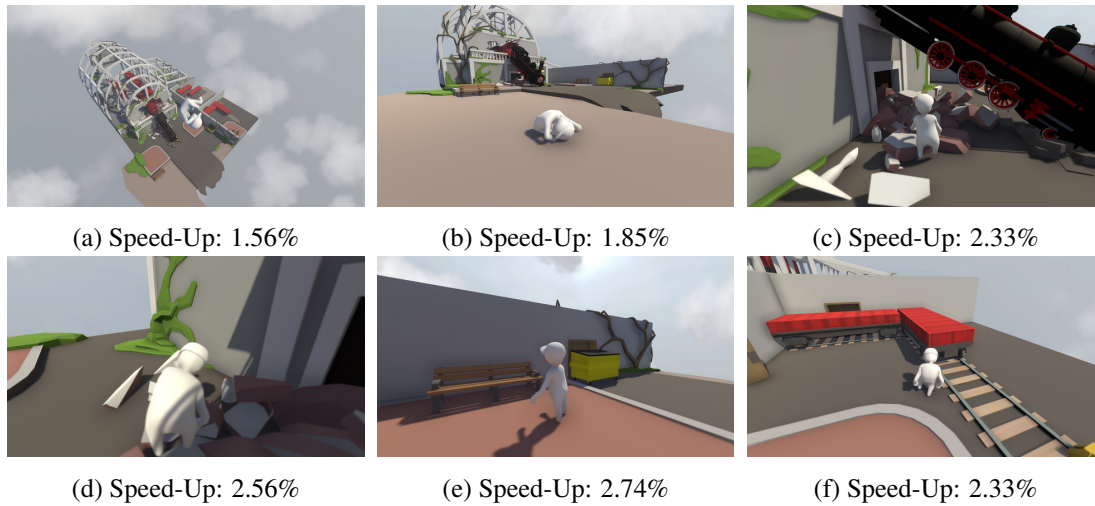


Figure 6.24: Human Fall Flat



Figure 6.25: Mount & Blade Warband



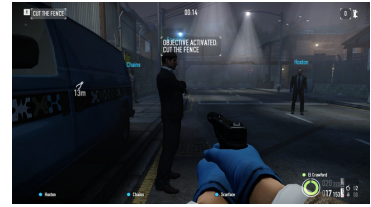
Figure 6.26: Overcooked 2



(a) Speed-Up: 0.17%



(b) Speed-Up: 0.29%



(c) Speed-Up: 0.59%



(d) Slow-Down: -1.41%



(e) Speed-Up: 0.45%



(f) Speed-Up: 0.36%

Figure 6.27: Payday 2



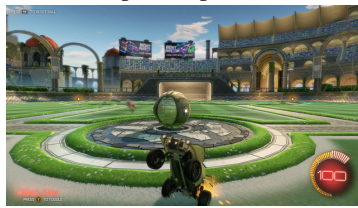
(a) Speed-Up: 0.94%



(b) Speed-Up: 0.99%



(c) Speed-Up: 1.08%



(d) Speed-Up: 1.37%



(e) Speed-Up: 1.02%



(f) Speed-Up: 1.34%

Figure 6.28: Rocket League



(a) Slow-Down: -1.59%



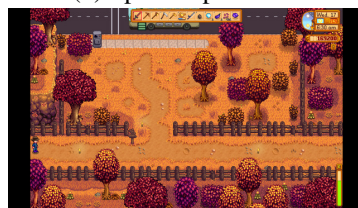
(b) Speed-Up: 0.71%



(c) Slow-Down: -2.02%



(d) Slow-Down: -0.59%



(e) Slow-Down: -0.94%



(f) Speed-Up: 0.53%

Figure 6.29: Stardew Valley

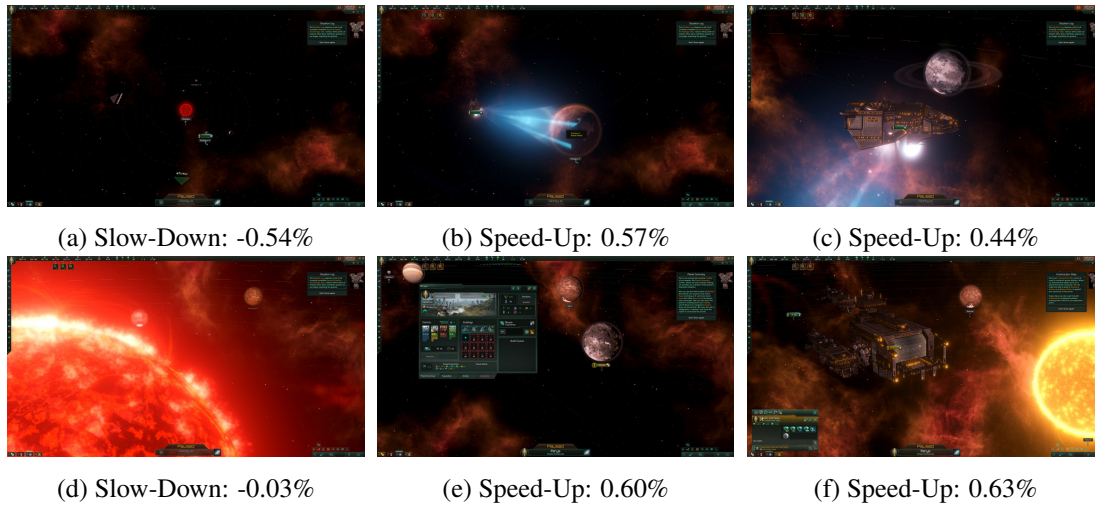


Figure 6.30: Stellaris

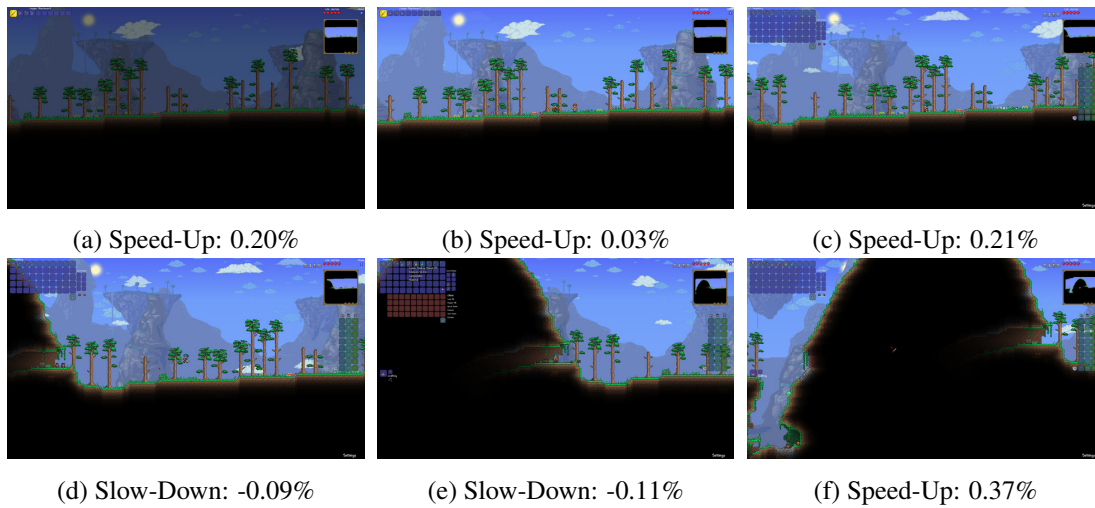


Figure 6.31: Terraria

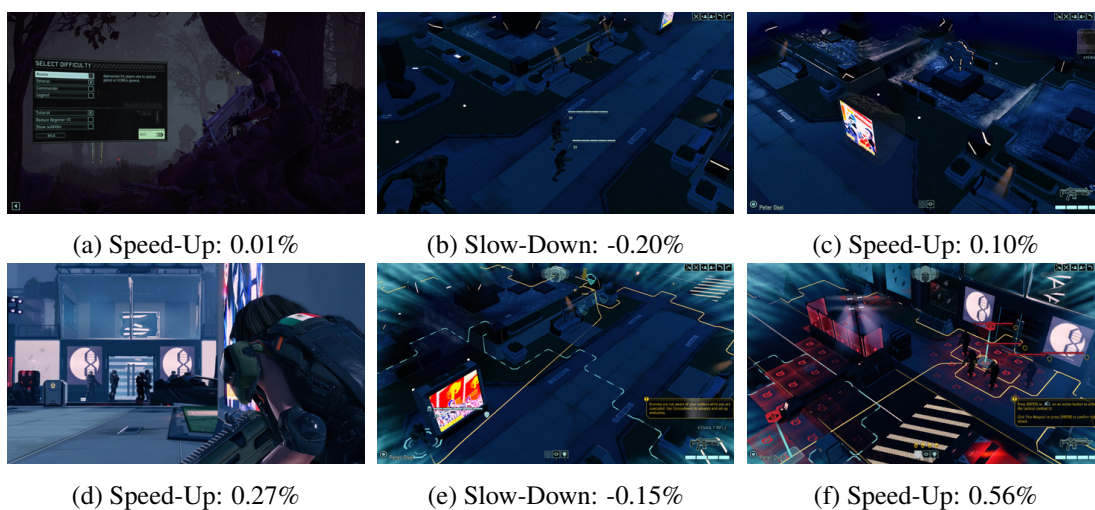


Figure 6.32: XCOM 2

These results are not universally positive, however. Simpler 2D games such as *Stardew Valley* and *Terraria* generally ran the same or slightly slower, with any reported speed-ups being $\leq 0.5\%$ (see [Figure 6.29](#) and [Figure 6.31](#)). The shaders and the uniform data used within these games are so simple that extra move instructions used to specialize partially constant uniforms, or artefacts of LunarGlass described in [Subsection 4.3.3](#) may negatively impact them more than the specializations can improve them.

In some 3D games such as *XCOM2*, the performance impact of specializing these shaders is also largely negligible (see [Figure 6.32](#)), with most speed-ups and slow-downs $\leq 0.5\%$. In *Pay Day 2*, minor speed-ups are present for several frames, but a performance drop of 1.4% occurs when viewing a close-up shot of a wire link fence, which may require a large alpha-blended texture (see [Figure 6.27](#)).

Even among Unity games, improvements are not guaranteed. In *Cities Skylines*, close-up zoomed-in scenes of buildings experience slow-downs of $\sim 1\%$, whereas zoomed-out scenes encompassing more birds-eye views of the city, which represent more typical game-play scenarios, demonstrate speed-ups of $\sim 3\%$ (see [Figure 6.21](#)). This demonstrates that even on a game engine where specialization based on constant uniforms generally provides benefits, there are renderings scenarios that can lead to performance regressions, so care must be taken to profile these effects.

These timing results show that it is not possible to predict the performance impact of specializing shaders based on the proportion of uniform data that is constant. The highest percentages of constant data in [Subsection 6.5.1](#) do not correspond to the largest speed-ups in [Figure 6.20](#). *Hollow Knight* has 80.94% constant uniforms, and experiences a smaller speed-up than *Euro Truck Simulator 2*, which has only 44.42% constant uniform data. Even within games using the same engine, or different types of scenes within the same games, the performance characteristics may vary drastically depending on the type of content being rendered and the shaders it requires.

6.6 Conclusion

By implementing apitrace trace analysis passes, this chapter re-affirms that large a portion of uniform data, including UBOs, is constant in a wide variety of popular commercial games. This constant uniform data was exported to LunarGlass, which ran a custom constant-folding pass on every shader before they were re-injected back into their original execution traces to time the performance difference. Using this tech-

nique, performance gains of up to 25.48% were achieved, with many other frames from different games experiencing 1% – 15% speed-ups, with worst-case-scenarios of only 1.5% slow-downs. Games using the popular Unity engine seemed particularly susceptible to performance improvements from specializing shaders with constant uniform data.

These results show that constant uniforms are available in the majority of commercial games, and for some of them, exploiting this data as part of a source-to-source pass can lead to noticeable reductions in rendering time in many frames. However, as shown in [Section 4.6](#), the results of any source-to-source shader optimization are not universally positive, so careful implementation and profiling of these techniques is required to ensure they speed up the game by exploiting constant data, rather than slow it down due to unwanted compilation artefacts.

Chapter 7

Conclusion

7.1 Summary

This research has explored numerous different techniques for automatically optimizing shaders in real-time graphics applications like computer games. Workloads from 25 popular commercial Linux-compatible games using the OpenGL graphics API were used to test these techniques, along with the well-known GFXBench 4.0 graphics benchmark. These workloads were characterized, subjected to numerous analysis passes, and used to test the efficacy of different optimizations across real-world games with a variety of complexities, game engines, and art-styles.

Beginning with an extensive background section in [Chapter 2](#), this thesis introduced the reader to the GPU accelerated graphics pipeline, and the shaders used to program it. This covered the basic concepts of 3D models made up of triangles, 2D texture images, and how parallel code running in vertex and fragment shaders determines how these objects appear on screen. The different methods for passing data from CPU to GPU, and between shader pipeline stages were also explained to allow code-motion optimizations and dataflow analysis in later chapters to be understood.

After introducing the reader to the modern programmable graphics pipeline, the evolution of this pipeline was explained in [Chapter 3](#). This chapter also explored work in adjacent fields of optimizing shaders for energy efficiency, and various shader simplification techniques that trade rendering accuracy for improved performance.

In [Chapter 4](#), source-to-source optimization of fragment shaders from GFXBench 4.0 was explored. These optimizations were performed using the LLVM-based LunarGlass offline compiler framework, which was extended to include several unsafe arithmetic optimizations. Iterative compilation was used to explore which optimiza-

tion passes were beneficial on a range of mobile and desktop GPUs from different vendors. Using a custom microbenchmarking tool designed to time small differences in fragment shader execution, it became clear that these optimization passes could have significant positive and negative effects on shader performance, and these results varied between different vendors. Loop unrolling, though seldom applicable to shaders, provided the most benefit in cases where it did apply. Unsafe floating-point arithmetic reassociation was widely applicable, and offered many small performance boosts. Hoisting code out of conditional statements had both positive and negative effects on code, so careful decisions needed to be made about when to apply it.

The work in [Chapter 5](#) expanded the scope of the previous chapter to explore full shader pipelines extracted from eight real-world games. Execution traces were recorded using the open-source apitrace tool, and LunarGlass was further extended to perform extensive static analysis on the extracted shader source code. This analysis included the detection of unused input data, constant-foldable code, and areas of code that could be moved to the CPU, or from the fragment to the vertex shader. Significant portions of code in conditionals could be extracted to the CPU, and many games declared large unused blocks of uniform data. An oracle study showed that these specializable portions of code could be greatly increased if it was known that certain types of input data were constant at run-time. By extending apitrace to analyse uniform input data, it was shown that all games had large percentages of uniform data that remained constant at run-time, so values close to the oracle study are likely achievable in practice. Timing tests were also performed to demonstrate the potential performance improvements possible by pruning unused uniform data from shaders in some cases.

Motivated by the prior chapter's analysis results, the work in [Chapter 6](#) demonstrates the performance improvements possible from exploiting the large portions of constant uniform data found in many games. This was achieved by further modifying apitrace to allow modification of all shaders used, and extending LunarGlass to allow constant-folding of uniform values that were detected by prior analysis of runtime traces. A further 17 games were selected from a wide variety of popular games with different engines, and these were demonstrated to have similar proportions of constant data as those examined previously. By exploiting this constant data across all shaders, the rendering cost of the entire frame could be improved by $\sim 5\%$ for many games with no degradation to rendering accuracy, with 10 – 25% improvements possible in some cases.

7.2 Critical Evaluation

Although some of the optimization and specialization techniques explored within this thesis gave promising results, the tools developed here remain as prototypes, and the analysis of them has several limitations described below.

Optimization Correctness Both the unsafe floating-point arithmetic optimizations in [Subsection 4.3.2](#) and the folding of constant uniforms into the shader source-code in [Chapter 6](#) have the potential to introduce slight differences due to altered floating-point precision. As graphics is a domain in which it is frequently preferable to trade slight computational inaccuracies in exchange for increased speed, these differences are likely to be acceptable in practice. When replaying the altered traces in [Chapter 6](#), any visual differences were identical to the naked eye. However, a more thorough evaluation of any differences in calculated pixel values would be beneficial before these approaches are deployed in a real system.

Guaranteeing Data Remains Constant One of the main limitations of the approaches in [Chapter 5](#) and [Chapter 6](#) for detecting which uniform data remains constant, is that only a limited number of frames are included in each trace. Although a variety of common gameplay scenarios are showcased throughout the several thousand frames in the trace, there is no guarantee that a uniform which is constant throughout the trace will be constant throughout the entire game.

One approach to mitigate this uncertainty would be to take multiple traces for each game to increase the coverage of different gameplay scenarios, and then take the intersection of all uniforms which remain constant between them. However, this does not remove the fundamental problem that it is almost impossible to determine which uniforms remain constant throughout an entire game, especially without access to the game's CPU-side source-code. The tools presented in this research are therefore only able to present a theoretical upper bound for the potential performance gains possible if all uniforms are known to be constant.

In their current state, these tools can only help indicate to developers where specialized variants of shaders would be useful for performance. Areas for future research may be to try and determine which key variables provide the most benefit to shaders if they are known to be constant. The initial findings in [Subsection 5.6.2](#) indicate that calculations involving combinations of only uniform and constant data are commonly

used as branch conditions. This could suggest that a few key uniforms may have disproportionate effects on the speed of the generated code if they are used to guard expensive branches, or as loop counters.

If it was determined which constant uniforms were most important, a specialized shader variant could be generated for the fast-path, and a generalized version could be used if these uniforms do not match the expected values. This type of system would require either modifications within the graphics driver, or within a game engine, so was deemed out of scope for this current research. However, without such an implementation, it is difficult to quantify whether overheads from this approach would make up for the potential performance gains demonstrated by this research.

Specialization vs Generalization The introduction of additional specialized shader variants may introduce overheads in several ways. Firstly, load-times would be increased due to additional shaders being compiled. Secondly, there may be context-switching overheads if there are multiple changes between the specialized and non-specialized variant of a shader. Finally, there are CPU-side performance costs for checking whether the expected set of constant uniforms match the current values within those variables in order to determine whether the specialized or generalized variant should be used. The costs of these overheads are difficult to quantify without implementing this system within a real engine or driver. One area of potential future research would be to determine not only which uniform variables are important to know the constant values of, but also which shaders are the most important to generate these specialized variants of. This research only provides an upper bound if all constant uniforms are exploited in all shaders.

Scope Limitations Throughout the 3 main experimental chapters, the scope has been progressively expanded. From individual fragment shaders from a single benchmark suite in [Chapter 4](#), to vertex-fragment shader pairs from 8 real games in [Chapter 5](#), these experiments culminated with timings from multiple full frames across 12 real games in [Chapter 6](#). Despite [Chapter 6](#)'s full-frame timings being representative of real rendering workloads, they do not capture the effect of the rest of the game's CPU-side code during real execution, so still do not perfectly measure the player-visible impact on performance that these optimizations would have. Additionally, these later experiments focused on a single desktop GPU on a Linux operating system. Future work may be possible to show how these results change between different vendors'

GPUs, or between different operating systems, or how different the impact on mobile vs desktop devices is.

Additional technical limitations also reduced the scope of experiments performed here. Several games were omitted as potential benchmarks here due to incompatibilities with apitrace. Other games were included in initial analysis of constant uniform/UBO data, but omitted from the final timing results due to incompatibilities with the LunarGlass-based compiler framework. Constant-folding modifications were only introduced for traditional uniforms, and not UBOs, in part due to these incompatibilities. This reduced the scope of the experiments, and prevented useful results from being generated about whether specializing shaders with UBOs produced similar benefits to those with traditional uniforms.

Another limitation was that the final tools were only able to substitute shaders which were linked into a single pipeline. Ideally, it should have been possible to generate different specialized variants of each shader depending on which pipeline program it was linked into, but these cases were ignored due to technical limitations. This may mean that some potential performance gains were missed by these experiments due to them not running on all shaders within the trace. It also prevented analysis of the trade-offs between using a single generalized shader versus multiple more specialized variants. Exploring the impacts of splitting shaders into multiple specialized variants, and specializing shaders based on UBO contents (which [Subsection 6.5.1](#) indicates have similar portions of constant data to traditional uniforms), would be areas that future work could expand on.

7.3 Directions for Future Work

The aim of this research is to motivate the development of further tools that will aid in automatically optimizing the performance of real-time graphics applications. The performance benefits demonstrated from exploiting constant uniform data shown in [Chapter 6](#) show that speed-ups are possible using only one of the potential optimization techniques described in [Chapter 5](#). Further work may seek to extend this performance analysis to capitalize on the potential of constant texture or vertex-buffer data, and the code-motion techniques that were applicable in many shaders.

This work focused on OpenGL, so future work into the newer low-level APIs like Vulkan or DirectX 12 may be interesting avenues of exploration. The new options in Vulkan for sending data from the CPU to GPU via specialization constants, or push-

constants may open up interesting opportunities. This research exploits only constant uniform data, but further analysis into moving small frequently updated uniforms to use push constants, or large blocks of seldom updated uniforms into separate uniform buffers may be additional areas that would be interesting to explore.

The programmable shaders examined at the heart of this research are part of the traditional vertex/fragment shader-based rasterization-based graphics pipeline, but newer options such as mesh shaders and real-time ray-tracing are emerging in modern graphics APIs. Most of the ideas presented in this research would also apply to these novel shader pipelines, but the dataflow patterns and performance characteristics would likely differ significantly. Exploring the applicability of the shader optimizations presented here to these new pipelines would be an intriguing direction for future work.

Perhaps the most practical direction for future work would be the implementation of these tools in a form that could be easily integrated into an overnight build system. If the process of extracting traces, performing analysis on them, and subsequently optimizing games using the techniques presented here could be fully automated, then the developer effort required to optimize games' performance could be reduced, yielding substantial benefits.

Bibliography

- [1] Kajal T Claypool and Mark Claypool. On frame rate and player performance in first person shooter games. *Multimedia systems*, 13(1):3–17, 2007.
- [2] Benjamin F Janzen and Robert J Teather. Is 60 fps better than 30?: the impact of frame rate and latency on moving target selection. In *CHI'14 Extended Abstracts on Human Factors in Computing Systems*, pages 1477–1482. ACM, 2014.
- [3] Research Nester. Computer graphics market : Global demand analysis & opportunity outlook 2024. <https://www.researchnester.com/reports/computer-graphics-market-global-demand-analysis-opportunity-outlook-2024/354>.
- [4] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 4.6). <https://www.opengl.org/registry/doc/glspec46.core.pdf>, 2019.
- [5] The Khronos Vulkan Working Group. Vulkan 1.0 Core API Specification. <https://www.khronos.org/registry/vulkan/specs/1.0/xhtml/vkspec.html>, 2016.
- [6] Microsoft. Direct3D 12 programming guide. [https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx), 2016.
- [7] German Ceballos, Andreas Sembrant, Trevor E Carlson, and David Black-Schaffer. Behind the scenes: Memory analysis of graphical workloads on tile-based gpus. In *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*, pages 1–11. IEEE, 2018.
- [8] Kishonti. GFXBench 4.0 - a benchmarking suite for OpenGL shaders. <https://gfxbench.com>.
- [9] Lewis Crawford and Michael O’Boyle. A cross-platform evaluation of graphics shader compiler optimization. In *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*, pages 219–228. IEEE, 2018.

- [10] LunarG. Lunarglass compiler stack. <https://lunarg.com/shader-compiler-technologies/lunarglass/>, 2011.
- [11] Lewis Crawford and Michael O’Boyle. Specialization opportunities in graphical workloads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 272–283. IEEE, 2019.
- [12] José Fonseca. apitrace graphics tracing tool. <http://apitrace.github.io/>, 2008.
- [13] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2019.
- [14] Kai Hormann, Konrad Polthier, and Alia Sheffer. Mesh parameterization: Theory and practice. In *ACM SIGGRAPH ASIA 2008 Courses*, SIGGRAPH Asia ’08. Association for Computing Machinery, 2008.
- [15] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.
- [16] Simon Green et al. Stupid OpenGL shader tricks. In *Advanced OpenGL Game Programming Course, Game Developers Conference*, 2003.
- [17] Fernando Navarro, Francisco J Serón, and Diego Gutierrez. Motion blur rendering: State of the art. In *Computer Graphics Forum*, volume 30, pages 3–26. Wiley Online Library, 2011.
- [18] Naty Hoffman. Color enhancement for videogames. In *SIGGRAPH Color Enhancement and Rendering in Film and Game Production course*, SIGGRAPH ’10. Association for Computing Machinery, 2010.
- [19] Louis Bavoil and Miguel Sainz. Screen space ambient occlusion. *NVIDIA developer information*: <http://developers.nvidia.com>, 6, 2008.
- [20] Open Signal. Android fragmentation visualized (august 2015). https://cdn.opensignal.com/public/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf, 2015.
- [21] John Leech. OpenGL ES Version 3.2. https://www.khronos.org/registry/OpenGL/specs/es/3.2/es_spec_3.2.pdf, 2019.

- [22] Khronos Group. About the khronos group. <https://www.khronos.org/about>, 2022.
- [23] Khronos Group. Khronos members. <https://www.khronos.org/members/list>, 2022.
- [24] Microsoft. DirectX-Specs. <https://microsoft.github.io/DirectX-Specs>, 2022.
- [25] Axon Samuel. The world's second-most popular desktop operating system isn't macos anymore. <https://arstechnica.com/gadgets/2021/02/the-worlds-second-most-popular-desktop-operating-system-isnt-macos-anymore>, 2021.
- [26] Valve. Steam hardware & software survey: December 2020. <https://web.archive.org/web/20210116081949/https://store.steampowered.com/hwsurvey>.
- [27] Ryan Smith. Apple deprecates opengl across all oses. <https://www.anandtech.com/show/12894/apple-deprecates-opengl-across-all-oses>, 2018.
- [28] Apple. Metal shading language specification version 1.2. <https://developer.apple.com/metal/metal-shading-language-specification.pdf>, 2016.
- [29] Tom Olson, Neil Trevett, Graham Sellers, John Kessenich, Jesse Barker, Laszlo Kishonti, Rys Sommefeldt, Slawek Grajewski, and Daniel Piers. More on vulkan and SPIR-V: The future of high-performance graphics. In *Game developers conference*, 2015.
- [30] Pawel Lapinski. Api without secrets: Introduction to vulkan* part 0: Preface. <https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-preface.html>, 2016.
- [31] Microsoft. High-level shader language (hlsl). <https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl>, 2021.
- [32] Tim Jones. Parsing direct3d shader bytecode. <http://timjones.io/blog/archive/2015/09/02/parsing-direct3d-shader-bytecode>, 2015.
- [33] Microsoft. DirectX intermediate language. <https://github.com/Microsoft/DirectXShaderCompiler/blob/main/docs/DXIL.rst>, 2022.
- [34] LLVM. LLVM language reference manual. <https://llvm.org/docs/LangRef.html>, 2022.

- [35] ISO/IEC. Working draft, standard for programming language c++. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4431.pdf>, 2015.
- [36] John Kessenich. OpenGL Shading Language 4.50 Specification. <https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf>, 2016.
- [37] John Kessenich, Boaz Ouriel, and Raun Krisch. Spir-v specification version 1.5, revision 4, unified. <https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf>, 2020.
- [38] Aras Pranckevičius. Cross platform shaders in 2014. <https://aras-p.info/blog/2014/03/28/cross-platform-shaders-in-2014>, 2014.
- [39] Nick Penwarden, Mathias Schott, and Evan Hart. Bringing unreal engine 4 to OpenGL. In *Game developers conference*, 2014.
- [40] Epic Games. HLSL cross compiler. <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Rendering/ShaderDevelopment/HLSLCrossCompiler>, 2022.
- [41] Rolando Caloca Olivares. ISV experience: Porting unreal engine 4 to vulkan. In *ACM SIGGRAPH 2016 Talks - Khronos 3D Graphics BoF: Vulkan, OpenGL, OpenGL ES*, 2016.
- [42] Axel Gneiting. ISV experience: Porting doom to vulkan. In *ACM SIGGRAPH 2016 Talks - Khronos 3D Graphics BoF: Vulkan, OpenGL, OpenGL ES*, 2016.
- [43] Dan Ginsburg. Performance results and lessons from porting source 2 to vulkan. In *Game developers conference - Vulkan Session Part II*, 2016.
- [44] Hans-Kristian Arntzen and Marius Bjørge. Porting to vulkan. In *Khronos Vulkan DevDay UK - Moving to Vulkan : How to make your 3D graphics more explicit*, 2016.
- [45] Adam Sawicki. Porting your engine to vulkan or DX12. In *Digital Dragons*, 2018.
- [46] Khronos Group. Vulkan gpu resources. <https://www.vulkan.org/tools>, 2022.
- [47] Khronos Group. OpenGL debugging tools. https://www.khronos.org/opengl/wiki/Debugging_Tools, 2021.

- [48] Khronos Group. OpenGL related toolkits and APIs. https://www.khronos.org/opengl/wiki/Related_toolkits_and_APIs, 2021.
- [49] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.
- [50] Ian Millington. *Game physics engine development: how to build a robust commercial-grade physics engine for your game*. CRC Press, 2010.
- [51] Christer Ericson. *Real-time collision detection*. Crc Press, 2004.
- [52] Guy Somberg. *Game Audio Programming: Principles and Practices*. CRC Press, 2017.
- [53] Eric Lengyel. *Foundations of Game Engine Development: Rendering*, volume 2. Terathon Software LLC., 2019.
- [54] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a pc game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, 1998.
- [55] Natalya Tatarchuk, Timothy Cooper, and Sebastian Aaltonen. Unity rendering architecture. In *ACM SIGGRAPH 2021 Talks - Rendering Engine Architecture Conference*, 2021.
- [56] Michael Vance. Rendering engine architecture at activision. In *ACM SIGGRAPH 2021 Talks - Rendering Engine Architecture Conference*, 2021.
- [57] Kim Byung-wook. Why develop in-house game engines? *The Korea Herald*, 2021.
- [58] Unreal engine - a 3d game engine and development environment. <https://www.unrealengine.com>, 2017.
- [59] Unity - a 3d game engine and development environment. <https://unity3d.com>, 2017.
- [60] Lars Doucet and Anthony Pecorella. Game engines on steam: The definitive breakdown. <https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown>, 2021.
- [61] Unity. Unity: Our impact by the numbers. <https://unity.com/our-company#key-facts>.

- [62] Unity plan comparison. <https://store.unity.com/compare-plans>, 2022.
- [63] Unity asset store. <https://assetstore.unity.com>, 2022.
- [64] Unity shader compilation. <https://docs.unity3d.com/Manual/shader-compilation.html>, 2022.
- [65] Unreal engine marketplace. <https://www.unrealengine.com/marketplace>, 2022.
- [66] Unreal engine licensing options. <https://www.unrealengine.com/license>, 2022.
- [67] Fortnite usage and revenue statistics (2022). <https://www.businessofapps.com/data/fortnite-statistics>, 2022.
- [68] Valve. Source engine. <https://developer.valvesoftware.com/wiki/Source>, 2022.
- [69] YoYo Games Ltd. Gamemaker. <https://gamemaker.io/en/gamemaker>, 2022.
- [70] Kadokawa Games. Rpg maker. <https://www.rpgmakerweb.com>, 2022.
- [71] Godot engine. <https://godotengine.org>, 2022.
- [72] OGRE engine. <https://www.ogre3d.org>, 2022.
- [73] LibGDX. <https://libgdx.com>, 2022.
- [74] Chris McClanahan. History and evolution of gpu architecture. *A Survey Paper*, 9:1–7, 2010.
- [75] Paul Jaquays and Brian Hook. Quake 3: Arena shader manual, revision 10. In *Game Developer’s Conference Hardcore Technical Seminar Notes*. Miller Freeman Game Group, 1999.
- [76] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 2.0). <https://registry.khronos.org/OpenGL/specs/gl/glspec20.pdf>, 2004.
- [77] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM transactions on graphics (TOG)*, 23(3):777–786, 2004.
- [78] Sarah Tariq. D3d11 tessellation. In *Game Developers Conference. Session: Advanced Visual Effects with Direct3D for PC*, 2009.

- [79] Mike Bailey. Gsl geometry shaders. *Oregon State University*, 2007.
- [80] Tianyun Ni. Direct compute: Bring gpu computing to the mainstream. In *GPU technology conference*, page 23. sn, 2009.
- [81] Hongly Va, Min-Hyung Choi, and Min Hong. Real-time cloth simulation using compute shader in unity3d for ar/vr contents. *Applied Sciences*, 11(17):8255, 2021.
- [82] Dody Dharma, Cliff Jonathan, A Imam Kistidjantoro, and Afwarman Manaf. Material point method based fluid simulation on gpu using compute shader. In *2017 International Conference on Advanced Informatics, Concepts, Theory, and Applications (ICAICTA)*, pages 1–6. IEEE, 2017.
- [83] Gareth Thomas. Compute-based gpu particle systems. In *Game Developers Conference, San Francisco, CA*, 2014.
- [84] Matthew Rusch, Neil Bickford, and Nuno Subtil. Introduction to vulkan ray tracing. In *Ray Tracing Gems II*, pages 213–255. Springer, 2021.
- [85] Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, and Roger Espasa. Shader performance analysis on a modern gpu architecture. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 10–pp. IEEE, 2005.
- [86] Victor Moya, Carlos González, Jordi Roca, Agustín Fernández, and Roger Espasa. A single (unified) shader gpu microarchitecture for embedded systems. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 286–301. Springer, 2005.
- [87] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.
- [88] Thanh Tuan Dao and Jaejin Lee. An auto-tuner for opencl work-group size on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):283–296, 2017.
- [89] Microsoft. D3D12 Shader Cache APIs. <https://microsoft.github.io/DirectX-Specs/d3d/ShaderCache.html>, 2022.

- [90] Xray Halperin, David Santiago, and Abdul Bezrati. Spider-man ig-impostors: cityscapes and beyond. In *SIGGRAPH Asia 2018 Technical Briefs*, pages 1–4. 2018.
- [91] Cem Cebenoyan. Stuttering in game graphics: Detection and solutions. In *China game developers conference*, 2012.
- [92] Fabian Giesen. A trip through the graphics pipeline 2011, part 1. <https://fgiesen.wordpress.com/2011/07/01/a-trip-through-the-graphics-pipeline-2011-part-1>, 2011.
- [93] Luis Valente, Aura Conci, and Bruno Feijó. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, volume 89, page 99. Citeseer, 2005.
- [94] Michael Worcester. Command buffers and pipelines. In *Khronos Vulkan DevDay UK - Moving to Vulkan : How to make your 3D graphics more explicit*, 2016.
- [95] Matthias Wloka. Batch, batch, batch: What does it really mean. In *Game developers conference*, 2003.
- [96] Cass Everitt, Tim Foley, John McDonald, and Graham Sellers. Approaching zero driver overhead in opengl. In *Game Developers Conference, San Francisco, CA*, 2014.
- [97] Diego Nehab, Joshua Barczak, and Pedro V Sander. Triangle order optimization for graphics hardware computation culling. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 207–211, 2006.
- [98] Jordy van Dortmont. Optimizing draw call batching using transient data-guided texture atlases. Master’s thesis, University of Utrecht, 2017.
- [99] Microsoft. Rendering from vertex and index buffers (direct3d 9). <https://docs.microsoft.com/en-us/windows/win32/direct3d9/rendering-from-vertex-and-index-buffers>, 2016.
- [100] Khronos Group. Opengl vertex specification. https://www.khronos.org/opengl/wiki/Vertex_Specification, 2022.

- [101] Khronos Group. Vertex specification best practices. https://www.khronos.org/opengl/wiki/Vertex_Specification_Best_Practices, 2018.
- [102] Khronos Group. Triangle primitives. https://www.khronos.org/opengl/wiki/Primitive#Triangle_primitives, 2020.
- [103] Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. Revisiting the vertex cache: Understanding and optimizing vertex processing on the modern gpu. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):1–16, 2018.
- [104] Dave Shreiner and Edward Angel. Interactive computer graphics: A top-down approach with shader-based opengl, 2012.
- [105] Jules Bloomenthal and Jon Rokne. Homogeneous coordinates. *The Visual Computer*, 11(1):15–26, 1994.
- [106] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O’Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 39–46, 2007.
- [107] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O’Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics (TOG)*, 27(4):1–23, 2008.
- [108] Kok-Lim Low. Perspective-correct interpolation. 2002.
- [109] The Khronos Group. OpenGL tessellation. <https://www.khronos.org/opengl/wiki/Tessellation>, 2020.
- [110] Ignacio Castaño. Watertight tessellation: precise and fma. <http://www.ludicon.com/castano/blog/2010/09/precise>, 2010.
- [111] Freddy Indra Wiryadi and Raymond Kosala. Particle rendering using geometry shader. In *2016 1st International Conference on Game, Game Art, and Gamification (ICGGAG)*, pages 1–6. IEEE, 2016.
- [112] Johan Andersson and Daniel Johansson. Shadows & decals: D3d10 techniques in frostbite (gdc’09). In *Game developers conference*, 2009.

- [113] Kai Lawonn. Computer graphics II - point shadows. https://vis.uni-jena.de/Lecture/ComputerGraphics2/Lec10_a_PointShadows.pdf, 2022.
- [114] Joey de Vries. Learn opengl - coordinate systems. <https://learnopengl.com/Getting-started/Coordinate-Systems>, 2020.
- [115] Fabian Giesen. Triangle rasterization in practice. <https://fgiesen.wordpress.com/2013/02/08/triangle-rasterization-in-practice>, 2013.
- [116] Wolfgang Straßer. *Schnelle Kurven und Flachendarstellung auf graphischen Sichtgeräten*. PhD thesis, Technischen Universität Berlin, 1974.
- [117] Edwin Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, The University of Utah, 1974.
- [118] Eugene Lapidous and Guofang Jiao. Optimal depth buffer for low-cost graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 67–73, 1999.
- [119] Microsoft. Stencil buffers. <https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/stencil-buffers>, 2019.
- [120] Shawn Hargreaves and Mark Harris. Deferred shading. In *Game Developers Conference*, volume 2, page 31, 2004.
- [121] The Khronos Group. Opengl blending. <https://www.khronos.org/opengl/wiki/Blending>, 2017.
- [122] The Khronos Group. OpenGL - uniform (GLSL). [https://www.khronos.org/opengl/wiki/Uniform_\(GLSL\)](https://www.khronos.org/opengl/wiki/Uniform_(GLSL)), 2020.
- [123] The Khronos Group. OpenGL - uniform buffer object. https://www.khronos.org/opengl/wiki/Uniform_Buffer_Object, 2017.
- [124] The Khronos Group. OpenGL - buffer object. https://www.khronos.org/opengl/wiki/Buffer_Object, 2021.
- [125] Adam Lake. Getting the most from opengl™ 1.2: How to increase performance by minimizing buffer copies on intel® processor graphics. <https://www.intel.com/content/www/us/en/developer/articles/training/getting-the-most-from-opengl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics.html>, 2014.

- [126] The Khronos Group. OpenGL - pixel transfer. https://www.khronos.org/opengl/wiki/Pixel_Transfer, 2020.
- [127] RasterGrid. Understanding GPU caches. <https://www.rastergrid.com/blog/gpu-tech/2021/01/understanding-gpu-caches>, 2021.
- [128] The Khronos Group. Program introspection: Uniforms and blocks. https://www.khronos.org/opengl/wiki/Program_Introspection#Uniforms_and_blocks, 2020.
- [129] The Khronos Group. OpenGL - interface block (GLSL). [https://www.khronos.org/opengl/wiki/Interface_Block_\(GLSL\)](https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)), 2021.
- [130] The Khronos Group. OpenGL - shader storage buffer object. https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object, 2020.
- [131] The Khronos Group. OpenGL - texture. <https://www.khronos.org/opengl/wiki/Texture>, 2020.
- [132] Waylon Brinck, Andrew Maximov, and Yibing Jiang. The technical art of uncharted 4. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, New York, NY, USA, 2016. Association for Computing Machinery.
- [133] The Khronos Group. OpenGL - sampler (GLSL). [https://www.khronos.org/opengl/wiki/Sampler_\(GLSL\)](https://www.khronos.org/opengl/wiki/Sampler_(GLSL)), 2020.
- [134] Turner Whitted and David M Weimer. A software test-bed for the development of 3-d raster graphics systems. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 271–277, 1981.
- [135] Turner Whitted and David M Weimer. A software testbed for the development of 3d raster graphics systems. *ACM Transactions on Graphics (TOG)*, 1(1):43–58, 1982.
- [136] Robert L Cook. Shade trees. *ACM Siggraph Computer Graphics*, 18(3):223–231, 1984.
- [137] Gregory D Abram and Turner Whitted. Building block shaders. *ACM SIGGRAPH Computer Graphics*, 24(4):283–288, 1990.
- [138] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

- [139] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298, 1990.
- [140] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. *ACM Siggraph computer graphics*, 22(4):21–30, 1988.
- [141] John Poulton. Pixel-planes: Building a vlsi-based graphics system. In *Proc. of 1985 Chapel Hill Conf. on VLSI*, pages 35–60, 1985.
- [142] John Eyles, John Austin, Henry Fuchs, Trey Greer, and John Poulton. Pixel-planes 4: A summary. *Advances in computer graphics hardware II*, pages 1833–207, 1988.
- [143] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. *ACM Siggraph Computer Graphics*, 23(3):79–88, 1989.
- [144] Brice Tebbs, Ulrich Neumann, John Eyles, Greg Turk, and David Ellsworth. Parallel architectures and algorithms for real-time synthesis of high quality images using deferred shading. Technical report, NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE, 1989.
- [145] Steven Molnar, John Eyles, and John Poulton. Pixelflow: high-speed rendering using image composition. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 231–240, 1992.
- [146] Henri Gouraud. Continuous shading of curved surfaces. *IEEE transactions on computers*, 100(6):623–629, 1971.
- [147] Gary Bishop and David M Weimer. Fast phong shading. *ACM SIGGRAPH Computer Graphics*, 20(4):103–106, 1986.
- [148] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann, and Amitabh Varshney. Real-time procedural textures. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 95–100, 1992.

- [149] Anselmo Lastra, Steven Molnar, Marc Olano, and Yulan Wang. Real-time programmable shading. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 59–ff, 1995.
- [150] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. Pixelflow: the realization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 57–68, 1997.
- [151] CORPORATE OpenGL Architecture ReviewBoard. *OpenGL reference manual: the official reference document for OpenGL, release 1*. Addison-Wesley Longman Publishing Co., Inc., 1992.
- [152] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: The pixelflow shading system. In *SIGGRAPH*, volume 98, pages 159–168, 1998.
- [153] Jon Leech. Opendgl extensions and restrictions for pixelflow. Technical Report TR98-019, Department of Computer Science, University of North Carolina, 1998.
- [154] Mark S Peercy, Marc Olano, John Airey, and P Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 425–432. ACM Press/Addison-Wesley Publishing Co., 2000.
- [155] Paul J Diefenbach and Norman I Badler. Multi-pass pipeline rendering: Realism for dynamic environments. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 59–ff, 1997.
- [156] Tom McReynolds, David Blythe, Brad Grantham, and Scott Nelson. Advanced graphics programming techniques using opengl. *Computer Graphics*, pages 95–145, 1998.
- [157] Erik Lindholm, Mark J Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM, 2001.
- [158] Harold Robert Feldman Zatz, Henry P Moreton, and John Erik Lindholm. Programmable pixel shading architecture, April 20 2004. US Patent 6,724,394.

- [159] Kekoa Proudfoot, William R Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 159–170. ACM, 2001.
- [160] William R Mark and Kekoa Proudfoot. Compiling to a vliw fragment pipeline. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 47–56, 2001.
- [161] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics (TOG)*, 22(3):896–907, 2003.
- [162] Martin Ecker. Programmable graphics pipeline architectures. *XEngine Corporation*, 2002.
- [163] Marc Olano, Kurt Akeley, John C Hart, Wolfgang Heidrich, Michael McCool, Jason L Mitchell, and Randi Rost. Real-time shading. In *ACM SIGGRAPH 2004 Course Notes*, pages 1–es. 2004.
- [164] Patrick S McCormick, Jeff Inman, James P Ahrens, Charles Hansen, and Greg Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *IEEE Visualization 2004*, pages 171–178. IEEE, 2004.
- [165] Nicolas Fritz, Philipp Lucas, and Philipp Slusallek. Cgis, a new language for data-parallel gpu programming. In *VMV*, pages 241–248, 2004.
- [166] Philipp Lucas, Nicolas Fritz, and Reinhard Wilhelm. The cgis compiler—a tool demonstration. In *International Conference on Compiler Construction*, pages 105–108. Springer, 2006.
- [167] Aaron Lefohn. Glift: An abstraction for generic, efficient gpu data structures. In *ACM SIGGRAPH 2005 Courses*, pages 140–es. 2005.
- [168] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *ACM SIGPLAN Notices*, 41(11):325–335, 2006.
- [169] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

- [170] NVIDIA. Nvidia cuda compute unified device architecture programming guide. 2007.
- [171] John E Stone, David Gohara, and Guochun Shi. Opencil: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.
- [172] Aaftab Munshi. The opencil specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.
- [173] Christoph Kubisch. Introduction to turing mesh shaders. *Nvidia Devblog*, 2018.
- [174] VV Sanzharov, AI Gorbonosov, VA Frolov, and AG Voloboy. Examination of the nvidia rtx. In *Proceedings of the 29th International Conference on Computer Graphics and Vision (GraphiCon 2019)*, volume 2485, page 7, 2019.
- [175] John Burgess. Rtx on—the nvidia turing gpu. *IEEE Micro*, 40(2):36–44, 2020.
- [176] Alexander Blake-Davies. Powering next-generation gaming visuals with amd rdna 2 and directx 12 ultimate. *AMD Community Blog*, 2020.
- [177] Microsoft. DirectX raytracing (dxr) functional spec v1.13. 2020.
- [178] Daniel Koch. Vulkan ray tracing final specification release. *Khronos Blog*, 2020.
- [179] Brian Guenter, Todd B Knoblock, and Erik Ruf. Specializing shaders. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 343–350. ACM, 1995.
- [180] David Luebke, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.
- [181] Jonathan Cohen, David Luebke, Nathaniel Duca, and Brenden Schubert. Glod: A geometric level of detail system at the opengl api level. In *IEEE Visualization*. Citeseer, 2003.
- [182] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, 1983.

- [183] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14. Eurographics Association, 2003.
- [184] Maryann Simmons and Dave Shreiner. Per-pixel smooth shader level of detail. In *ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1. ACM, 2003.
- [185] Fabio Pellacini. User-configurable automatic shader simplification. *ACM Transactions on Graphics (TOG)*, 24(3):445–452, 2005.
- [186] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG)*, 30(6):1–12, 2011.
- [187] Lutz Kettner. Fast automatic level of detail for physically-based materials. In *ACM SIGGRAPH 2017 Talks*, page 39. ACM, 2017.
- [188] Chang-Woo Cho, Chung-Pyo Hong, Jin-Chun Piao, Yeong-Kyu Lim, and Shin-Dug Kim. Performance optimization of 3d applications by opengl es library hooking in mobile devices. In *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, pages 471–476. IEEE, 2014.
- [189] Jin-Chun Piao, Chang-Woo Cho, Cheong-Ghil Kim, Bernd Burgstaller, and Shin-Dug Kim. An adaptive lod setting methodology with opengl es library on mobile devices. In *2014 International Conference on IT Convergence and Security (ICITCS)*, pages 1–4. IEEE, 2014.
- [190] Rui Wang, Xianjin Yang, Yazhen Yuan, Wei Chen, Kavita Bala, and Hujun Bao. Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics (TOG)*, 33(6):226, 2014.
- [191] Diego Nehab, Pedro V Sander, Jason Lawrence, Natalya Tatarchuk, and John R Isidoro. Accelerating real-time shading with reverse reprojection caching. In *Graphics hardware*, volume 41, pages 61–62, 2007.
- [192] Pitchaya Sitthi-amorn, Jason Lawrence, Lei Yang, Pedro V Sander, Diego Nehab, and Jiahe Xi. Automated reprojection-based pixel shader optimization. In *ACM SIGGRAPH Asia 2008 papers*, pages 1–11. 2008.

- [193] Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. A system for rapid, automatic shader level-of-detail. *ACM Transactions on Graphics (TOG)*, 34(6):187, 2015.
- [194] Yong He, Tim Foley, and Kayvon Fatahalian. A system for rapid exploration of shader optimization choices. *ACM Transactions on Graphics (TOG)*, 35(4):112, 2016.
- [195] Yazhen Yuan, Rui Wang, Tianlei Hu, and Hujun Bao. Runtime shader simplification via instant search in reduced optimization space. In *Computer Graphics Forum*, volume 37, pages 143–154. Wiley Online Library, 2018.
- [196] FE Allen. Program optimization, annual review of automatic programming, vol. 5, 1969.
- [197] John T Bagwell Jr. Local optimizations. In *Proceedings of a symposium on Compiler optimization*, pages 52–66, 1970.
- [198] Thomas J. Watson IBM Research Center. Research Division, FE Allen, and J Cocke. *A catalogue of optimizing transformations*. 1971.
- [199] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques, and tools. *Addison wesley*, 7(8):9, 1986.
- [200] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.
- [201] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.
- [202] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.
- [203] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementation. *ACM SIGPLAN Notices*, 28(6):90–99, 1993.
- [204] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

- [205] Jaejin Lee, Samuel P Midkiff, and David A Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 114–130. Springer, 1997.
- [206] Jens Knoop. Parallel constant propagation. In *European Conference on Parallel Processing*, pages 445–455. Springer, 1998.
- [207] Edward M Riseman and Caxton C Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, 100(12):1405–1411, 1972.
- [208] DW Anderson, FJ Sparacio, and Robert M Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.
- [209] Ivan Flores. Lookahead control in the ibm system 370 model 165. *Computer*, 7(11):24–38, 1974.
- [210] Roland N Ibbett. The mu5 instruction pipeline. *The Computer Journal*, 15(1):42–50, 1972.
- [211] JE Smith. A study of branch prediction techniques. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–147, 1981.
- [212] Sparsh Mittal. A survey of techniques for dynamic branch prediction. *Concurrency and Computation: Practice and Experience*, 31(1):e4666, 2019.
- [213] Shien-Tai Pan, Kimming So, and Joseph T Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 76–84, 1992.
- [214] Cliff Young, Nicolas Gloy, and Michael D Smith. A comparative analysis of schemes for correlated branch prediction. *ACM SIGARCH Computer Architecture News*, 23(2):276–286, 1995.
- [215] Marius Evers, Sanjay J Patel, Robert S Chappell, and Yale N Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, 1998.

- [216] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.
- [217] I-Cheng K Chen, John T Coffey, and Trevor N Mudge. Analysis of branch prediction via data compression. *ACM SIGPLAN Notices*, 31(9):128–137, 1996.
- [218] John Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984.
- [219] Pierre Michaud. A ppm-like, tag-based branch predictor. *Journal of Instruction Level Parallelism*, 7(1):1–10, 2005.
- [220] Hongliang Gao and Huiyang Zhou. Pmpm: Prediction by combining multiple partial matches. *Journal of Instruction-Level Parallelism*, 9:1–18, 2007.
- [221] André Seznec. A case for (partially)-tagged geometric history length predictors. *Journal of InstructionLevel Parallelism*, 2006.
- [222] André Seznec. Analysis of the o-geometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 394–405. IEEE, 2005.
- [223] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127, 2011.
- [224] André Seznec. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9:1–6, 2007.
- [225] André Seznec. A 64 kbytes isl-tage branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.
- [226] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. Branch classification: a new mechanism for improving branch predictor performance. *International Journal of Parallel Programming*, 24(2):133–158, 1996.
- [227] Gabriel H Loh and Dana S Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings. International Conference on Parallel Architectures and Compilation Techniques*, pages 165–176. IEEE, 2002.

- [228] Ayose Falcón, Jared Stark, Alex Ramirez, Konrad Lai, and Mateo Valero. Prophet/critic hybrid branch prediction. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 250–261. IEEE, 2004.
- [229] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.
- [230] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.
- [231] Samuel P Harbison. An architectural alternative to optimizing compilers. *ACM SIGPLAN Notices*, 17(4):57–65, 1982.
- [232] Stephen E Richardson. *Caching function results: Faster arithmetic by avoiding unnecessary computation*. Sun Microsystems Laboratories, 1992.
- [233] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. Value locality and load value prediction. *ACM SIGPLAN Notices*, 31(9):138–147, 1996.
- [234] Mikko H Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 226–237. IEEE, 1996.
- [235] Freddy Gabbay and Avi Mendelson. Speculative execution based on value prediction. Technical Report 1080, Technion-IIT, Department of Electrical Engineering, 1996.
- [236] Freddy Gabbay and Abraham Mendelson. *An Experimental and Analytical Study of Speculative Execution based on Value Prediction*. Technion-IIT, Department of Electrical Engineering, 1997.
- [237] Rubén González, Adrián Cristal, Daniel Ortega, Alexander Veidenbaum, and Mateo Valero. A content aware integer register file organization. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 314–324. IEEE, 2004.

- [238] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load value approximation. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 127–139. IEEE, 2014.
- [239] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Speculative execution on gpu: An exploratory study. In *2010 39th International Conference on Parallel Processing*, pages 453–461. IEEE, 2010.
- [240] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Value prediction and speculative execution on gpu. *International Journal of Parallel Programming*, 39(5):533–552, 2011.
- [241] Enqiang Sun and David Kaeli. Aggressive value prediction on a gpu. *International Journal of Parallel Programming*, 42(1):30–48, 2014.
- [242] Haonan Wang, Mohamed Ibrahim, Sparsh Mittal, and Adwait Jog. Address-stride assisted approximate load value prediction in gpus. In *Proceedings of the ACM International Conference on Supercomputing*, pages 184–194, 2019.
- [243] Amir Yazdanbakhsh, Gennady Pekhimenko, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. Towards breaking the memory bandwidth wall using approximate value prediction. In *Approximate Circuits*, pages 417–441. Springer, 2019.
- [244] Sparsh Mittal. A survey of value prediction techniques for leveraging value locality. *Concurrency and computation: practice and experience*, 29(21):e4250, 2017.
- [245] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.
- [246] Gilbert J Hansen. Adaptive systems for the dynamic run-time optimization of programs. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1974.
- [247] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, L Mitchell Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, H Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, 1957.

- [248] JL Dawson. Combining interpretive code with machine code. *The Computer Journal*, 16(3):216–219, 1973.
- [249] RJ Dakin and Peter C Poole. A mixed code approach. *The Computer Journal*, 16(3):219–222, 1973.
- [250] PJ Brown. Throw-away compiling. *Software: Practice and Experience*, 6(3):423–434, 1976.
- [251] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, 1997.
- [252] Niklaus Wirth. The programming language oberon. *Software: Practice and Experience*, 18(7):661–670, 1988.
- [253] Thomas Peter Kistler. *Continuous program optimization*. University of California, Irvine, 1999.
- [254] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, 2001.
- [255] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.
- [256] Philip Samuel Abrams. An apl machine. Technical report, Stanford Linear Accelerator Center, Calif., 1970.
- [257] Terrence C Miller. Tentative compilation: A design for an apl compiler. *ACM SIGAPL APL Quote Quad*, 9(4-P1):88–95, 1979.
- [258] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
- [259] L Peter Deutsch and Allan M Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, 1984.
- [260] David Ungar and Randall B Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, 1987.

- [261] Urs Hölzle. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, 1994.
- [262] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, 1994.
- [263] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *Ieee micro*, 17(3):36–43, 1997.
- [264] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):1–32, 2008.
- [265] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10, 2013.
- [266] Armin Rigo. Representation-based just-in-time specialization and the psycho prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, 2004.
- [267] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 781–796, 2014.
- [268] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80, 2009.
- [269] Jungwoo Ha, Mohammad R Haghighat, Shengnan Cong, and Kathryn S McKinley. A concurrent trace-based just-in-time compiler for single-threaded

- javascript. In *Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures*, pages 47–54. Citeseer, 2009.
- [270] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. *ACM SIGPLAN Notices*, 47(6):239–250, 2012.
- [271] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.
- [272] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 191–201, 1989.
- [273] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 314–321, 1995.
- [274] Peter Sestoft and Alexandre V. Zamulin. Annotated bibliography on partial evaluation and mixed computation. *New Generation Computing*, 6(2&3):309–354, 1988.
- [275] David Keppel, Susan J Eggers, and Robert R Henry. *A case for runtime code generation*. Department of Computer Science and Engineering, University of Washington, 1991.
- [276] David Keppel, Susan J Eggers, and Robert R Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, 1993.
- [277] Rob Pike, Bart Locanthi, and John Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software: Practice and Experience*, 15(2):131–151, 1985.
- [278] Bart N Locanthi. Fast bitblt () with asm () and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, 1987.
- [279] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

- [280] Peter Holst Andersen. Partial evaluation applied to ray tracing. In *Software Engineering in Scientific Computing*, pages 78–85. Springer, 1996.
- [281] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück. Fortran program specialization. *ACM SIGPLAN Notices*, 30(4):61–70, 1995.
- [282] Romana Baier, Robert Glück, and Robert Zöchling. Partial evaluation of numerical programs in fortran. *PEPM*, 94:119–132, 1994.
- [283] Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization*, pages 137–146. Springer, 1996.
- [284] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation*, pages 54–72. Springer, 1996.
- [285] Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 145–156, 1996.
- [286] François Noël, Luke Hornof, Charles Consel, and Julia L Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*, pages 132–142. IEEE, 1998.
- [287] Todd B Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 215–225, 1996.
- [288] Charles Consel, Luke Hornof, Renaud Marlet, Gilles Muller, Scott Thibault, E-N Volanschi, Julia Lawall, and Jacques Noyé. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys (CSUR)*, 30(3es):19–es, 1998.
- [289] Charles Consel, Julia L Lawall, and Anne-Françoise Le Meur. A tour of tempo: A program specializer for the c language. *Science of Computer Programming*, 52(1-3):341–370, 2004.
- [290] GCC, the GNU Compiler Collection. <https://gcc.gnu.org>.

- [291] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasice, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS)*, 19(2):217–251, 2001.
- [292] Gilles Muller, Renaud Marlet, E-N Volanschi, Charles Consel, Calton Pu, and Ashvin Goel. Fast, optimized sun rpc using automatic program specialization. In *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No. 98CB36183)*, pages 240–249. IEEE, 1998.
- [293] Dawson R Engler, Wilson C Hsieh, and M Frans Kaashoek. C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 131–144, 1996.
- [294] Massimiliano Poletto, Wilson C Hsieh, Dawson R Engler, and M Frans Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–369, 1999.
- [295] Christopher W Fraser and David R Hanson. A code generation interface for ansi c. *Software: Practice and Experience*, 21(9):963–988, 1991.
- [296] Christopher W Fraser and David R Hanson. *A retargetable C compiler: design and implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [297] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J Eggers. An evaluation of staged run-time optimizations in dyc. *ACM SIGPLAN Notices*, 34(5):293–304, 1999.
- [298] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248(1-2):147–199, 2000.
- [299] P Geoffrey Lowney, Stefan M Freudenberger, Thomas J Karzes, WD Lichtenstein, Robert P Nix, John S O’donnell, and John C Ruttenberg. The multi-flow trace scheduling compiler. In *Instruction-Level Parallelism*, pages 51–142. Springer, 1993.

- [300] Mark Leone and Peter Lee. Lightweight run-time code generation. *PEPM*, 94:97–106, 1994.
- [301] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. *ACM SIGPLAN Notices*, 31(5):137–148, 1996.
- [302] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [303] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.
- [304] Chris A Lattner. *Macroscopic data structure analysis and optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.
- [305] Alexis Engelke and Martin Schulz. Robust practical binary optimization at run-time using llvm. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 56–64. IEEE, 2020.
- [306] Alexis Engelke and Martin Schulz. Instrew: Leveraging llvm for high performance dynamic binary instrumentation. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 172–184, 2020.
- [307] Josef Weidendorfer and Jens Breitbart. The case for binary rewriting at runtime for efficient implementation of high-level programming models in hpc. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 376–385. IEEE, 2016.
- [308] Alexis Engelke and Josef Weidendorfer. Using llvm for optimized lightweight binary re-writing at runtime. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 785–794. IEEE, 2017.
- [309] Vasil Vasilev, Ph Canal, Axel Naumann, and Paul Russo. Cling—the new interactive interpreter for root 6. In *Journal of Physics: Conference Series*, volume 396, page 052071. IOP Publishing, 2012.

- [310] Thibaut Lutz and Vinod Grover. Lambdajit: a dynamic compiler for heterogeneous optimizations of stl algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, pages 99–108, 2014.
- [311] Henri-Pierre Charles and Victor Lomüller. Is dynamic compilation possible for embedded systems? In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 80–83, 2015.
- [312] Victor Lomüller. *Générateur de code multi-temps et optimisation de code multi-objectifs*. PhD thesis, Université de Grenoble, 2014.
- [313] Kavon Farvardin, H Finkel, M Kruse, and J Reppy. atjit: A just-in-time auto-tuning compiler for c++. In *LLVM Developer’s Meeting Technical Talk*, 2018.
- [314] Hal Finkel, David Poliakoff, Jean-Sylvain Camier, and David F Richards. Clangjit: Enhancing c++ with just-in-time compilation. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 82–95. IEEE, 2019.
- [315] David Keppel. *Runtime code generation*. PhD thesis, University of Washington, 1996.
- [316] Tito Autrey and Michael Wolfe. Initial results for glacial variable analysis. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 120–134. Springer, 1996.
- [317] Ulrik Jørring and William L Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 86–96, 1986.
- [318] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 259–269. IEEE, 1997.
- [319] Brad Calder, Peter Feller, Alan Eustace, et al. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1(1):1–6, 1999.
- [320] Freddy Gabbay and Avi Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 270–280. IEEE Computer Society, 1997.

- [321] Robert Muth, Scott Watterson, and Saumya Debray. Code specialization based on value profiles. In *International Static Analysis Symposium*, pages 340–359. Springer, 2000.
- [322] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. Energy efficient source code transformation based on value profiling. In *Proc. International Workshop on Compilers and Operating Systems for Low Power*, 2000.
- [323] Scott Watterson and Saumya Debray. Goal-directed value profiling. In *International Conference on Compiler Construction*, pages 319–333. Springer, 2001.
- [324] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *International Symposium on Code Generation and Optimization (CGO’07)*, pages 198–208. IEEE, 2007.
- [325] Minhaj Ahmad Khan. Improving performance through deep value profiling and specialization with code transformation. *Computer Languages, Systems & Structures*, 37(4):193–203, 2011.
- [326] Taewook Oh, Hanjun Kim, Nick P Johnson, Jae W Lee, and David I August. Practical automatic loop specialization. *ACM SIGARCH Computer Architecture News*, 41(1):419–430, 2013.
- [327] Sylvain Henry, Hugo Bolloré, and Emmanuel Oseret. Towards the generalization of value profiling for high-performance application optimization. Technical report, Exascale Computing Research Laboratory, Campus Teratec, 2015.
- [328] Cédric Valensi. A generic approach to the definition of low-level components for multi-architecture binary analysis. *Université de Versailles-St Quentin en Yvelines*, 2014.
- [329] Cédric Valensi. Madras: Multi-architecture binary rewriting tool. Technical report, University of Versailles Saint-Quentin en Yvelines, 2013.
- [330] Shasha Wen, Milind Chabbi, and Xu Liu. Redspy: Exploring value locality in software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 47–61, 2017.

- [331] Daniel Wong, Nam Sung Kim, and Murali Annavaram. Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 176–187. IEEE, 2016.
- [332] Sylvain Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in gpgpu computations. In *European Conference on Parallel Processing*, pages 46–55. Springer, 2009.
- [333] Ram Rangan, Mark W Stephenson, Aditya Ukarande, Shyam Murthy, Virat Agarwal, and Marc Blackstein. Zeroploit: Exploiting zero valued operands in interactive gaming applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(3):1–26, 2020.
- [334] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O’Connor, and Stephen W Keckler. Flexible software profiling of gpu architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 185–197. IEEE, 2015.
- [335] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. Gvprof: a value profiler for gpu-based clusters. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1263–1278. IEEE Computer Society, 2020.
- [336] Guilherme Vieira Leobas and Fernando Magno Quintão Pereira. Semiring optimizations: dynamic elision of expressions with identity and absorbing elements. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.
- [337] Mark Stephenson and Ram Rangan. Pgz: automatic zero-value code specialization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 36–46, 2021.
- [338] Mark W Stephenson and Ram Rangan. Azp: Automatic specialization for zero values in gaming applications. *arXiv preprint arXiv:2011.10550*, 2020.
- [339] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zero and data reuse-aware fast convolution for deep neural networks on gpu. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2016.

- [340] Kihwan Choi, Karthik Dantu, Wei-Chung Cheng, and Massoud Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 732–737, 2002.
- [341] Zhijian Lu, John Lach, Mircea Stan, and Kevin Skadron. Reducing multimedia decode power using feedback control. In *Proceedings 21st International Conference on Computer Design*, pages 489–496. IEEE, 2003.
- [342] Christopher J Hughes and Sarita V Adve. A formal approach to frequent energy adaptations for multimedia applications. *ACM SIGARCH Computer Architecture News*, 32(2):138, 2004.
- [343] Chaeseok Im, Soonhoi Ha, and Huiseok Kim. Dynamic voltage scheduling with buffers in low-power multimedia applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4):686–705, 2004.
- [344] Gauthier Lafruit, Lode Nachtergaele, Kristof Denolf, and Jan Bormans. 3d computational graceful degradation. In *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 547–550. IEEE, 2000.
- [345] Nicolaas Tack, Francisco Morán, Gauthier Lafruit, and Rudy Lauwereins. 3d graphics rendering time modeling and control for mobile terminals. In *Proceedings of the ninth international conference on 3D Web technology*, pages 109–117, 2004.
- [346] Bren Mochocki, Kanishka Lahiri, and Srihari Cadambi. Power analysis of mobile 3d graphics. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 1–6. IEEE, 2006.
- [347] Bren C Mochocki, Kanishka Lahiri, Srihari Cadambi, and X Sharon Hu. Signature-based workload estimation for mobile 3d graphics. In *Proceedings of the 43rd annual design automation conference*, pages 592–597, 2006.
- [348] Yan Gu, Samarjit Chakraborty, and Wei Tsang Ooi. Games are up for dvfs. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 598–603. IEEE, 2006.

- [349] Yan Gu and Samarjit Chakraborty. Power management of interactive 3d games using frame structures. In *21st International Conference on VLSI Design (VLSI-D 2008)*, pages 679–684. IEEE, 2008.
- [350] Yan Gu and Samarjit Chakraborty. Control theory-based dvs for interactive 3d games. In *2008 45th ACM/IEEE Design Automation Conference*, pages 740–745. IEEE, 2008.
- [351] Yan Gu and Samarjit Chakraborty. A hybrid dvs scheme for interactive 3d games. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–12. IEEE, 2008.
- [352] Da-Jing Zhang-Jian, Chung-Nan Lee, Ing-Jer Huang, and Shiann-Rong Kuang. Power estimation for interactive 3d game using an efficient hierarchical-based frame workload prediction. In *Proceedings: APSIPA ASC 2009: Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference*, pages 208–215. Asia-Pacific Signal and Information Processing Association, 2009 Annual . . . , 2009.
- [353] Benedikt Dietrich, Swaroop Nunna, Dip Goswami, Samarjit Chakraborty, and Matthias Gries. Lms-based low-complexity game workload prediction for dvfs. In *2010 IEEE International Conference on Computer Design*, pages 417–424. IEEE, 2010.
- [354] Tulika Mitra and Tzi-cker Chiueh. Dynamic 3d graphics workload characterization and the architectural implications. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 62–71. IEEE, 1999.
- [355] Glenn Deen, Matthew Hammer, John Bethencourt, Iris Eiron, John Thomas, and James H Kaufman. Running quake II on a grid. *IBM Systems Journal*, 45(1):21–44, 2006.
- [356] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI*, volume 6, pages 12–12, 2006.
- [357] John E Laird. Using a computer game to develop advanced ai. *Computer*, 34(7):70–75, 2001.

- [358] Christian Bauckhage, Christian Thureau, and Gerhard Sagerer. Learning human-like opponent behavior for interactive computer games. In *Joint Pattern Recognition Symposium*, pages 148–155. Springer, 2003.
- [359] Tye Hooley, Burt Hunking, Mike Henry, and Atsushi Inoue. Generation of emotional behavior for non-player characters-development of emobot for quake II. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 954–955. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2004.
- [360] Jarkko M Valtjus-Anttila, Timo Koskela, and Seamus Hickey. Power consumption model of a mobile gpu based on rendering complexity. In *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 210–215. IEEE, 2013.
- [361] Benedikt Dietrich and Samarjit Chakraborty. Managing power for closed-source android os games by lightweight graphics instrumentation. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–3. IEEE, 2012.
- [362] Benedikt Dietrich and Samarjit Chakraborty. Lightweight graphics instrumentation for game state-specific power management in android. *Multimedia Systems*, 20(5):563–578, 2014.
- [363] Xiaohan Ma, Zhigang Deng, Mian Dong, and Lin Zhong. Characterizing the performance and power consumption of 3d mobile games. *Computer*, 46(4):76–82, 2012.
- [364] Beilei Sun, Xi Li, Jiachen Song, Zhinan Cheng, Yuan Xu, and Xuehai Zhou. Texture-directed mobile gpu power management for closed-source games. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pages 348–354. IEEE, 2014.
- [365] Benedikt Dietrich and Samarjit Chakraborty. Power management using game state detection on android smartphones. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 493–494, 2013.

- [366] Benedikt Dietrich and Samarjit Chakraborty. Forget the battery, let's play games! In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 1–8. IEEE, 2014.
- [367] Zhinan Cheng, Xi Li, Beilei Sun, Ce Gao, and Jiachen Song. Automatic frame rate-based dvfs of game. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 158–159. IEEE, 2015.
- [368] Iman Soltani Mohammadi, Mohammad Ghanbari, and Mahmoud Reza Hashemi. Gamorra: An api-level workload model for rasterization-based graphics pipeline architecture. *Computers & Graphics*, 2022.
- [369] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.
- [370] Jurn-Gyu Park, Chen-Ying Hsieh, Nikil Dutt, and Sung-Soo Lim. Quality-aware mobile graphics workload characterization for energy-efficient dvfs design. In *2014 IEEE 12th symposium on embedded systems for real-time multimedia (ESTIMedia)*, pages 70–79. IEEE, 2014.
- [371] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous mpsoes. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.
- [372] Chen-Ying Hsieh, Jurn-Gyu Park, Nikil Dutt, and Sung-Soo Lim. Memory-aware cooperative cpu-gpu dvfs governor for mobile games. In *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, pages 1–8. IEEE, 2015.
- [373] Sparsh Mittal and Jeffrey S Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23, 2014.
- [374] Daecheol You and K-S Chung. Dynamic voltage and frequency scaling framework for low-power embedded gpus. *Electronics letters*, 48(21):1333–1334, 2012.

- [375] BVN Silpa, Gummidipudi Krishnaiah, and Preeti Ranjan Panda. Rank based dynamic voltage and frequency scaling fortified graphics processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 3–12, 2010.
- [376] Po-Han Wang, Yen-Ming Chen, Chia-Lin Yang, and Yu-Jung Cheng. A predictive shutdown technique for gpu shader processors. *IEEE Computer Architecture Letters*, 8(1):9–12, 2009.
- [377] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power gating strategies on gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(3):1–25, 2011.
- [378] Kent W Nixon, Xiang Chen, Hucheng Zhou, Yunxin Liu, and Yiran Chen. Mobile gpu power consumption reduction via dynamic resolution and frame rate scaling. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*, 2014.
- [379] Yu Yan, Songtao He, Yunxin Liu, and Longbo Huang. Optimizing power consumption of mobile games. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, pages 21–25, 2015.
- [380] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Parallel frame rendering: Trading responsiveness for energy on a mobile gpu. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 83–92. IEEE, 2013.
- [381] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. *ACM SIGARCH Computer Architecture News*, 42(3):529–540, 2014.
- [382] Georgios Keramidas, Chrysa Kokkala, and Iakovos Stamoulis. Clumsy value cache: An approximate memoization technique for mobile gpu fragment shaders. In *Workshop on Approximate Computing (WAPCO'15)*, 2015.
- [383] Jon McCaffrey. Exploring mobile vs. desktop opengl performance 24. *OpenGL Insights*, 337:341, 2012.

- [384] Iosif Antochi, Ben Juurlink, Stamatis Vassiliadis, and Petri Liuha. Memory bandwidth requirements of tile-based rendering. In *International Workshop on Embedded Computer Systems*, pages 323–332. Springer, 2004.
- [385] Thomas J Olson. Hardware 3d graphics acceleration for mobile devices. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5344–5347. IEEE, 2008.
- [386] Eric Haines and Steven Worley. Fast, low memory z-buffering when performing medium-quality rendering. *journal of graphics tools*, 1(3):1–5, 1996.
- [387] Enrique De Lucas, Pedro Marcuello, Joan-Manuel Parcerisa, and Antonio Gonzalez. Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):473–485, 2018.
- [388] Xuejun Hao and Amitabh Varshney. Variable-precision rendering. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 149–158, 2001.
- [389] Ju-Ho Sohn, Ramchan Woo, and Hoi-Jun Yoo. A programmable vertex shader with fixed-point simd datapath for low power wireless applications. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 107–114, 2004.
- [390] Jeevan Chittamuru, Wayne Burleson, and Jeongseon Euh. Dynamic wordlength variation for low-power 3d graphics texture mapping. In *2003 IEEE Workshop on Signal Processing Systems (IEEE Cat. No. 03TH8682)*, pages 251–256. IEEE, 2003.
- [391] Kurt Akeley and Jonathan Su. Minimum triangle separation for correct z-buffer occlusion. In *Graphics Hardware*, volume 10, pages 1283900–1283904. New York, NY, USA, 2006.
- [392] Jeff Pool, Anselmo Lastra, and Montek Singh. Energy-precision tradeoffs in mobile graphics processing units. In *2008 IEEE International Conference on Computer Design*, pages 60–67. IEEE, 2008.
- [393] Jeff Pool, Anselmo Lastra, and Montek Singh. Precision selection for energy-efficient pixel shaders. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 159–168, 2011.

- [394] Jeff Pool. *Energy-precision tradeoffs in the graphics pipeline*. PhD thesis, University of North Carolina, 2012.
- [395] Jeff Pool, Anselmo Lastra, and Montek Singh. Power-gated arithmetic circuits for energy-precision tradeoffs in mobile graphics processing units. *Journal of Low Power Electronics*, 7(2):148–162, 2011.
- [396] Slo-Li Chu, Chih-Chieh Hsiao, and Chen-Yu Chen. A dual-mode unified shader with frame-based dynamic precision adjustment for mobile gpus. In *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*, pages 158–165. IEEE, 2011.
- [397] Chih-Chieh Hsiao, Slo-Li Chu, and Chen-Yu Chen. Energy-aware hybrid precision selection framework for mobile gpus. *Computers & Graphics*, 37(5):431–444, 2013.
- [398] Slo-Li Chu, Chih-Chieh Hsiao, and Chen-Yu Chen. Program-based dynamic precision selection framework with a dual-mode unified shader for mobile gpus. *Computers & Electrical Engineering*, 39(7):2183–2196, 2013.
- [399] NVIDIA. Nvidia nsight graphics. <https://developer.nvidia.com/nsight-graphics>.
- [400] AMD. Radeon developer tools suite. <https://gpuopen.com/tools>.
- [401] Intel. Intel graphics performance analyzers. <https://software.intel.com/content/www/us/en/develop/tools/graphics-performance-analyzers.html>.
- [402] Sheng Guo, Philipp Gerasimov, and Bonnie Aona. Practical game performance analysis using intel graphics performance analyzers. *Intel Corporation White Paper*, 2011.
- [403] Arm. Arm graphics analyzer. <https://developer.arm.com/tools-and-software/embedded/arm-development-studio/components/graphics-analyzer>.
- [404] Qualcomm. Snapdragon debugger. <https://developer.qualcomm.com/software/snapdragon-debugger-visual-studio>.
- [405] Qualcomm. Snapdragon profiler. <https://developer.qualcomm.com/software/snapdragon-profiler>.

- [406] Apple. Using metal system trace in instruments to profile your app. https://developer.apple.com/documentation/metal/using_metal_system_trace_in_instruments_to_profile_your_app.
- [407] Apple. Frame capture debugging tools. https://developer.apple.com/documentation/metal/frame_capture_debugging_tools.
- [408] Microsoft. Pix - performance tuning and debugging for directx 12 games on windows. <https://devblogs.microsoft.com/pix>, 2017.
- [409] Matthew Fisher. GPUView. <https://graphics.stanford.edu/~mdfisher/GPUView.html>.
- [410] Microsoft. Using GPUView. <https://docs.microsoft.com/en-us/windows-hardware/drivers/display/using-gpuview>.
- [411] Michael Sartain. GPUVis. <https://github.com/mikesart/gpuvis>, 2017.
- [412] Baldur Karlsson. Renderdoc - a stand-alone vulkan, d3d11, d3d12, and opengl graphics debugging tool. <https://renderdoc.org/>, 2012.
- [413] LunarG. vktrace capture/replay tool. <https://github.com/LunarG/vktrace>, 2016.
- [414] LunarG. GFXReconstruct - tools for the capture and replay of vulkan api calls. <https://github.com/LunarG/gfxreconstruct>, 2018.
- [415] Mark Friedell, Mark LaPolla, Sandeep Kochhar, Steve Sistare, and Janusz Juda. Visualizing the behavior of massively parallel programs. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 472–480, 1991.
- [416] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. Data visualization and performance analysis in the prism programming environment. In *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, pages 37–52, 1992.
- [417] Nathaniel Duca, Krzysztof Niski, Jonathan Bilodeau, Matthew Bolitho, Yuan Chen, and Jonathan Cohen. A relational debugging engine for the graphics pipeline. *ACM Transactions on Graphics (TOG)*, 24(3):453–463, 2005.
- [418] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM transactions on graphics (TOG)*, 21(3):693–702, 2002.

- [419] Magnus Strengert, Thomas Klein, and Thomas Ertl. A hardware-aware debugger for the opengl shading language. In *Graphics Hardware*, volume 9, 2007.
- [420] Ahmad Sharif and Hsien-Hsin S Lee. Total recall: a debugging framework for gpus. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 13–20, 2008.
- [421] Bryce Van Dyk, Christof Lutteroth, Gerald Weber, and Burkhard Wünsche. *Using opengl state history for graphics debugging*. Václav Skala-UNION Agency, 2013.
- [422] Qiming Hou, Kun Zhou, and Baining Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. In *ACM SIGGRAPH Asia 2009 papers*, pages 1–11. 2009.
- [423] Michael Wimmer and Peter Wonka. Rendering time estimation for real-time rendering. In *Rendering Techniques*, pages 118–129, 2003.
- [424] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. IEEE, 2009.
- [425] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. Cudaadvisor: LLVM-based runtime profiling for modern gpus. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 214–227, 2018.
- [426] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.
- [427] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.
- [428] Khronos Group. glslang - khronos-reference front end for glsl/essl, partial front end for hlsl, and a spir-v generator. <https://github.com/KhronosGroup/glslang>.

- [429] Khronos Group. Spirv-cross - a tool designed for parsing and converting spir-v to other shader languages. <https://github.com/KhronosGroup/SPIRV-Cross>.
- [430] Greg Spencer, Peter Shirley, Kurt Zimmerman, and Donald P Greenberg. Physically-based glare effects for digital images. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 325–334, 1995.
- [431] Guennadi Riguer, Natalya Tatarchuk, and John Isidoro. Real-time depth of field simulation. *ShaderX2: Shader Programming Tips and Tricks with DirectX*, 9:529–556, 2004.
- [432] Erik Reinhard, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski. *High dynamic range imaging: acquisition, display, and image-based lighting*. Morgan Kaufmann, 2010.
- [433] Gary McTaggart, Chris Green, and Jason Mitchell. High dynamic range rendering in valve’s source engine. In *ACM SIGGRAPH 2006 Courses*, pages 7–es. 2006.
- [434] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.
- [435] Matthias Hullin, Elmar Eisemann, Hans-Peter Seidel, and Sungkil Lee. Physically-based real-time lens flare rendering. In *ACM SIGGRAPH 2011 papers*, pages 1–10. 2011.
- [436] Sungkil Lee and Elmar Eisemann. Practical real-time lens-flare rendering. In *Computer Graphics Forum*, volume 32, pages 1–6. Wiley Online Library, 2013.
- [437] Henry Moreton. Watertight tessellation using forward differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 25–32, 2001.
- [438] The mesa 3d graphics library. <https://www.mesa3d.org/>.
- [439] Morgan McGuire. The supershader. *Shader X4: advanced rendering techniques*, pages 485–498, 2005.
- [440] ARM. Mali offline compiler. <https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/mali-offline-compiler>.

- [441] Bill Bilodeau. Advanced visual effects with directx 11: Vertex shader tricks - new ways to use the vertex shader to improve performance. In *Game developers conference*, 2014.
- [442] Michal Drobot. GCN Execution Patterns in Full Screen Passes. <https://michaldrobot.com/2014/04/01/gcn-execution-patterns-in-full-screen-passes>, 2014.
- [443] August Ferdinand Möbius. *Der barycentrische Calcul, ein Hülfsmittel zur analytischen Behandlung der Geometrie (etc.)*. Barth, 1827.
- [444] Piers Daniell. ARB_timer_query - Extension specification. http://developer.download.nvidia.com/opengl/specs/GL_ARB_timer_query.txt, 2009.
- [445] Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 3.3 (core profile)). <https://www.khronos.org/registry/OpenGL/specs/gl/glspec33.core.pdf>, 2010.
- [446] Maurice Ribble. EXT_disjoint_timer_query - Extension specification. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_disjoint_timer_query.txt, 2013.
- [447] Arthur Zuckerman. 75 steam statistics: 2020/2021 facts, market share & data analysis. <https://comparecamp.com/steam-statistics/#TOC1>, 2020.
- [448] Martin Benjamins and Pavel Djundik. Steam database. <https://steamdb.info>.
- [449] Google. Snappy, a fast compressor/decompressor. <https://github.com/google/snappy>.
- [450] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.