Heriot-Watt University

4th Year Dissertation

Implementation of a Vulkan-based renderer in Processing

Author:
Téo Taylor

Supervisor:
Mr. Rob Stewart

A thesis submitted in fulfilment of the requirements for the degree of MSc.

in the

School of Mathematical and Computer Sciences

March 2025

Declaration of Authorship

I, Téo Taylor, declare that this thesis titled, "Implementation of a Vulkan-based renderer in Processing" and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date: 27/03/25

Abstract

"Processing" is a framework which includes a minimalistic IDE and provides functionality for rendering animated visual effects. Processing's hardware-accelerated 3D renderer uses an API called OpenGL to render graphics on the GPU. However, OpenGL is an older API that has performance limitations compared to newer APIs. Vulkan is the modern successor to OpenGL, and related work shows that it can achieve higher throughput in rendering performance, and make better use of hardware resources. This paper implements a new Vukan renderer into Processing, and then compares the OpenGL and Vulkan renderers across 13 rendering benchmarks to assess the performance improvements. Hardware profiling tools are also used to identify what causes the performance gains with the Vulkan renderer. The paper concludes that Vulkan is faster than OpenGL in almost all benchmarks. Additionally, this paper detail the scenarios in which the Vulkan renderer outperforms OpenGL, and the reasons for those performance differences using the hardware profiling data. Processing might gain from Vulkan's speed advantages. These findings would benefit the Processing community, which includes a wide range of programmers from beginner to advanced levels. This paper also discusses Processing's ongoing efforts to modernise the platform, whilst fulfilling the community's requirement for backwards compatibility; a newer, faster API would help contribute to this modernisation effort, but there are trade-offs between performance and backwards-compatibilty, and this paper provides insights into future work to achieve a balance of both.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my supervisor, Rob Stewart, who has provided unwavering support, and invaluable guidance throughout this project. He has expressed genuine interest in my work and the Processing project, while providing professional feedback, which has helped immensely in my endeavours. His clever suggestions, such as using Intel VTune, has resulted in me being able to achieve conclusions and insights that I never imagined I could achieve from the beginning of the project.

Next, I would like to express my gratitude and appreciation for the open-source Processing project, and the talented individuals who continue to maintain it out of love for computational art. For the past 6 years, Processing has been my platform for learning coding and creating personal projects. Its minimalist, yet flexible environment has helped to solidify my programming skills, and progress in my university studies. I would also like to express gratitude towards Raphaël de Courville and Stef Tervelde for the interest in my project and the opportunity to discuss Processing's future; they have provided invaluable insights for the conclusion in this paper. I certainly wish to contribute to the modernisation efforts very soon.

Not least of all, my deepest appreciation goes towards towards my family; my parents and my sister who have been the beacon of support and encouragement throughout my university years. Their presence and caringness has helped me push through challenging times, and I will be forever grateful.

Contents

List of Figures

List of Tables

Abbreviations

IDE - **I**ntegrated **D**evelopment **E**nvironment

CPU - **C**entral **P**rocessing **U**nit

GPU - **G**raphical **P**rocessing **U**nit

UMA - **U**nified **M**emory **A**ccess

API - **A**pplication **P**rogramming **I**nterface

OS - **O**perating **S**ystem

Chapter 1

Introduction

1.1 Context

There is a piece of software named "Processing" [Reas and Fry, 2007] which provides a minimalistic IDE and a simplified language for creating programs with animated visuals. It is an efficient way for programmers, regardless of skill level, to write small programs in Processing. These programs are called "sketches". The programmer writes code into the IDE, then clicks a play button. This starts up the programmer's sketch, which displays the programmer's animated visuals.

The framework features many functions to render various graphics on to the screen. To render these graphics, Processing features a hardware accelerated renderer - this uses the GPU to render graphics. To interface with the GPU, the renderer uses a popular API called OpenGL [Ope, 2025], which is a cross-platform graphics API that abstracts the GPU hardware from the programmer. OpenGL processes the programmer's commands on the CPU, and sends data to the GPU for the scene to be rendered. This takes place in the background in Processing and this is not normally visible to the Processing developer.

However, as GPU hardware continued to evolve, OpenGL's old design caused drivers to have a large overhead [Shiraef, 2016]; that is, the CPU and GPU used a lot of its computing resources performing tasks such as handing backwards compatibility, error-checking (to notify the programmer of any programming errors), and supporting legacy features.

These limitations prompted the creation of newer graphics APIs, and in 2016, the maintainers of OpenGL known as The Kronos Group, released the successor to OpenGL; Vulkan [Vul, 2025b]. Vulkan was designed to address the limitations of OpenGL and be the new industry-leading standard for cross-platform computer graphics [Lujan et al., 2019]. This, in essence, improves performance compared to an application that uses OpenGL. This paper explores the possibility of a Vulkan-based renderer in Processing.

1.2 Aim

In this project, a new Vulkan-based renderer is developed, and tested against Processing's OpenGL renderer using a set of test programs in the form of Processing sketches. The primary objective of the project is to determine the performance differences between the OpenGL renderer and the Vulkan renderer.

1.3 Objectives

1. (O1) To find out **what** the overall average difference in performance across all sketches when comparing OpenGL and Vulkan.
2. (O2) To develop a new Vulkan renderer for the Processing framework.
3. (O3) To create 13 benchmarks that will compare the performance of the Vulkan renderer against the OpenGL renderer.
4. (O4) To find out **what** the time taken is to process each frame in the sketch's runtime in OpenGL and Vulkan.
5. (O5) To find out **what** the performance difference is between OpenGL and Vulkan when using Processing's immediate and retained modes (more details about these modes can be found in Chapter 3).
6. (O6) To find out **how** the sketch utilises the hardware in OpenGL and Vulkan, by monitoring CPU usage information, GPU usage information, and memory usage information using Intel VTune.
7. (O7) To find out **why** the sketches perform the way they do in OpenGL and Vulkan, by using the findings from Intel VTune to link it to the performance results.

Throughout this report, the objectives will be referenced in the form of O1, O2, O3, O4, O5, O6 and O7.

Chapter 2

Background

2.1 Processing

2.1.1 Processing Overview

Processing is described as a flexible software sketchbook for learning, prototyping, and creative coding through visual arts [Pro, 2025]. The language used in the application, known as the “Processing language”, is designed to generate and modify images, with an objective to achieve a balance between clarity and advanced features [Reas and Fry, 2007].

While the Processing language is known to be its own programming language, it is in reality a Java dialect which gets preprocessed into pure Java code [Linares-Pellicer et al., 2009]. When the user clicks on the ”play” button to begin a sketch, Processing performs preprocessing operations in the background, and runs the Java program, without exposing the pure Java code to the user.

Every compiled sketch is a Java program that includes Processing’s framework code, called the Processing core. A Processing sketch contains programmer’s code inside a class, that of which extends the main Processing core class called PApplet [Fry, 2008].

2.2 OpenGL

2.2.1 OpenGL's history

The following section will provide some history and context behind OpenGL, which explains its evolution up to today with its current design.

Silicon Graphics, Inc, or SGI, was the company that formulated OpenGL. OpenGL was initially developed to tackle issues from SGI's "IRIS GL" API, which was difficult to port to other systems (such as Microsoft's DOS or Windows). The OpenGL Architecture Review Board was formed and the initial specification for OpenGL 1.0 was completed on June 30th, 1992 [Kilgard and Akeley, 2008, Sellers et al., 2013]. OpenGL was structured as a state machine; objects were created and processed based on OpenGL's current state settings [Kilgard, 1997]. The "state machine" design philosophy remained throughout OpenGL's lifetime, even through major design overhauls. One such major design change was in OpenGL 2.0 in 2004, integrating programmable GPU pipelines into the formerly fixed-function pipeline, which meant that graphics programming was much more flexible. [Wolff, 2011]. In 2006, OpenGL development was handed over to the Khronos group [Amador Cambronero et al., 2021].

To maintain backwards-compatibility, OpenGL also retained much of its old designs dating back to version 1.0, and integrated new features alongside the legacy features. However, this proved problematic as the older features were incompatible with modern GPUs, and version 3.0 introduced a deprecation model in an effort to gradually phase out older features [Shreiner et al., 2009].

OpenGL's development from this point forward involved adding new features, but it became clear around this time that OpenGL's fundamental structure was flawed, and an entirely new system would be needed [Kenwright, 2017]. Rather than overhauling the existing system and working around its limitations, the Khronos group superseded OpenGL with Vulkan in 2016 [Tolo et al., 2018]. However, OpenGL continues to exist to this day as a usable graphics API, albeit with some performance limitations.

2.2.2 OpenGL's performance issues

OpenGL is a mature API that is still used by a wide range of applications today. However, as technology evolved, the following limitations became apparent;

- **Driver overhead -** OpenGL has a large overhead [Campbell, 2023, Shiraef, 2016], because the specification has design limitations that makes translating graphical workloads to modern hardware complex; this requires the CPU and GPU to use a lot of computational resources.
- **Lack of multi-threading support -** OpenGL was not designed to take advantage of multiple cores in the CPU, which poses a serious problem in today's world of multi-core devices [Kenwright, 2017]. This is caused by the state machine model; OpenGL has one global state, and multiple threads trying to access this one state will result in race conditions, unless properly managed by thread-safety mechanisms, such as mutexes or locks [Szabó and Illés, 2022]. Even then, attempting a thread-safe multithreaded approach with OpenGL will worsen performance rather than improved it, because of thread synchronisation, lack of parallelism (only one thread could run at a time), and the driver performing context switching [Nott, 2013].
- **Error checking -** OpenGL performs error checking to inform programmers of errors in their OpenGL code; this uses CPU time [Lapinski, 2017]. This is not ideal as error checking is performed regardless of the program being in a development state or a final build.
- **Discontinuation and eventual obsolescence -** OpenGL is no longer being developed by The Khronos Group, and it will eventually become a deprecated API on all platforms [Szabó and Illés, 2020]. Additionally, it will also increasingly lack support for new hardware features, such as ray-tracing [Allison et al., 2024, Unterguggenberger et al., 2023]. Currently, significant effort is being put into modernising the latest revision of Processing [Fry, 2021], so the deprecation of OpenGL may hold back Processing in that regard.

2.3 Vulkan

2.3.1 Vulkan overview

This project explores Vulkan as a potential solution to the problem of the high OpenGL overhead. Vulkan is a relatively new API which has been designed to address the limitations of OpenGL. It is designed in such a way that the developer has full control over what Vulkan does; this elimates unexpected behaviour due to OpenGL's abstraction of

graphics hardware [Lujan et al., 2021]. The Vulkan API is very verbose, and operates on a much lower level than OpenGL, which provides the programmer with much more control over the hardware [Unterguggenberger et al., 2022].

Vulkan addresses the following limitations from OpenGL:

- **Driver overhead -** Vulkan is designed to represent modern hardware much better than OpenGL; this reduces driver overhead significantly [Chandra, 2018]. Additionally, the programmer must be very explicit about how they want their GPU hardware to run; this further reduces overhead as less automation and translation is needed from the driver.
- **Multi-threading support -** Vulkan is designed with multithreading in mind; this eliminates the single-core limitations that OpenGL is faced with [Bodurri, 2019]. This is done by having multiple state objects rather than a single global state; each thread can perform on its own state object.
- **Error checking -** Vulkan eliminates the overhead of error checking in non-development builds through using a feature called "validation layers"; an extension which reports programming errors is loaded during testing. When the programmer exports a non-development build of the program, the validation layers are removed and no error checking is performed, which completely eliminates unnecessary error-checking overheads [Pankratz et al., 2021].

As a consequence of the verbosity and low-level design, Vulkan is significantly more complicated to program than OpenGL; a typical OpenGL program takes 150 lines of code to render a single triangle; Vulkan takes around 1000 lines of code to do the same. [Unterguggenberger et al., 2023].

2.3.2 Performance improvements from other works

A study from Lujan et al. has shown that Vulkan had significant improvements in overall performance over OpenGL. This was because OpenGL took signficiantly more single-core CPU time which severely bottlenecked the GPU's workload, while Vulkan used multiple threads to distribute the workload. Nilssen has shown that Vulkan can be up to 96% faster than OpenGL. Other studies from Ferraz et al. and Lujan et al. have shown that Vulkan saves substantial energy which is especially beneficial for battery-powered devices like smartphones. Processing sketches can also run on smartphones using Processing's "Android Mode" [Colubri, 2023], therefore Android Processing users could benefit from a Vulkan implementation of the renderer too.

2.4 Related work

2.4.1 P5.js

P5.js is a web-based alternative to Processing which runs in a web browser using Javascript [Sandberg, 2019]. It is very similar to Processing (on desktop) and has a similar WebGL mode, which is effectively the web-based version of OpenGL. Technically, modern web browsers such as Google Chrome on Windows use a translation layer called Google ANGLE. ANGLE (on Google Chrome, Windows) translates WebGL commands to Microsoft's Direct3D 9 commands for improved compatibility [Donaldson et al., 2017], which is similar to OpenGL. Contributors of ANGLE, Kligge have also performed benchmarks comparing native OpenGL to ANGLE translating OpenGL ES to Direct3D 9, and found that ANGLE performs on-par to native OpenGL. This indicates that P5.js has the same performance as Processing's OpenGL renderer.

Results from Varanka's work seems to suggest that Direct3D 9 is faster, though this is highly dependant on the environment and there is no further evidence to suggest that P5.js's WebGL renderer is faster than Processing's OpenGL renderer. It is also worth noting that WebGL must abide to the same design basis as OpenGL, and therefore inherits its limitations; that is: lack of multithreading and higher overhead.

2.4.2 OpenFrameworks

OpenFrameworks is a framework for creative coding, similar to Processing, but uses C++ instead of Java [Chibalashvili et al., 2023], and has OpenGL support [OFo, 2025]. There is a project by Gfrerer which implements Vulkan into OpenFrameworks, though the Vulkan renderer is designed around "[embracing] the "no-surprises" policy" in such a way that integrates with the framework, meaning that the programmer is fully exposed to Vulkan and must program much of the Vulkan code from scratch, rather than being a drop-in replacement for the OpenGL renderer.

Chapter 3

Project design

3.1 Preface

This chapter discusses the design details of the Vulkan renderer used in this project, named "PVXD". Processing's OpenGL-based renderer is called P2D/P3D, but for simplicity, it will be referred to as "PXD".

3.2 Processing's design

3.2.1 PGL

Processing (and, by extension, PXD) is designed in a modular way, such that its components can be replaced by others. One module of PXD is an OpenGL abstraction layer called PGL. Using a custom renderer, Processing's modularity can be taken advantage of, and OpenGL commands can be re-directed to alternative code via PGL.

3.2.2 Immediate and retained mode

Processing is designed to be highly flexible with simple code, which is difficult to implement with the rigidity of graphics cards. The overall end goal is to assemble vertex information on the CPU and buffer this data to the GPU. There are two main ways

Processing renders graphics; by using *immediate* and *retained* mode [Colubri and Fry, 2012];

Immediate: In this mode, Processing calculates vertex data and then sends this to the GPU *every frame*. Once the GPU has received vertex information, a draw command is sent to draw the vertex data. This approach is typically slower than retained mode, but much more flexible, and easier to code.

However, sending a draw command for each piece of geometry drawn is inefficient when using OpenGL; hence, Processing uses a batching process to batch all the commands into a buffer of a specified size. Once this size limit is reached, Processing will send the buffer to the GPU and issue a draw command. The process then repeats until all geometry in the frame has been drawn.

Retained: In this mode, Processing calculates vertex data and sends this to the GPU *only once*. This data is then drawn every frame. This is typically faster than immediate mode as it avoids expensive per-frame GPU buffering routines, at the expense of flexibility, making it slightly more involved to program. The buffering process is typically done at the sketch's startup, using a class called "PShape" which enables the programmer to define geometry to be rendered using retained mode.

There is an exception to this rule however. The programmer can specify child PShape objects and geometry, which can be independantly transformed (moving, scaling, rotating, etc). Doing this will cause Processing to re-calculate the vertex data and re-buffer this to the GPU; a process very similar to immediate mode, and hence is slower than retained mode with static geometry. For clarity, using retained mode with moving (dynamic) geometry is referred to as *dynamic retained* whilst using retained mode with non-moving (static) geometry is referred to as *static retained*.

A simple way to avoid this is to move the base PShape object (without any child objects) using the transform() function, then call shape() every time the programmer wants the object to be rendered. This incurs more OpenGL calls which, as previously discussed in Chapter 2, has a high overhead associated to it. However, this should be less than the overhead of calculating and buffering vertex data on the CPU, as it offloads vertex calculations to the GPU and avoids unnecessary buffering. This technique is used in the

project's evaluation, and for clarity, this is referred to as *optimised retained*. As this can be rather confusing to keep in mind, this table provides clarity on the different modes:

TABLE 3.1: Rendering modes

Mode	Speed	Flexibility	Static or dynamic
Immediate	Slower	Flexible	Both
Static retained	Faster	Inflexible	Static
Dynamic retained	Slower	Inflexible	Dynamic
Optimised retained	Faster	Inflexible	Dynamic

3.3 Design plans

This section outlines the design plans for the Vulkan renderer, which contains a package called "GL2VK". GL2VK houses 3 major components that is described in the following subsections. A high-level view of the renderer is shown in figure 3.1.

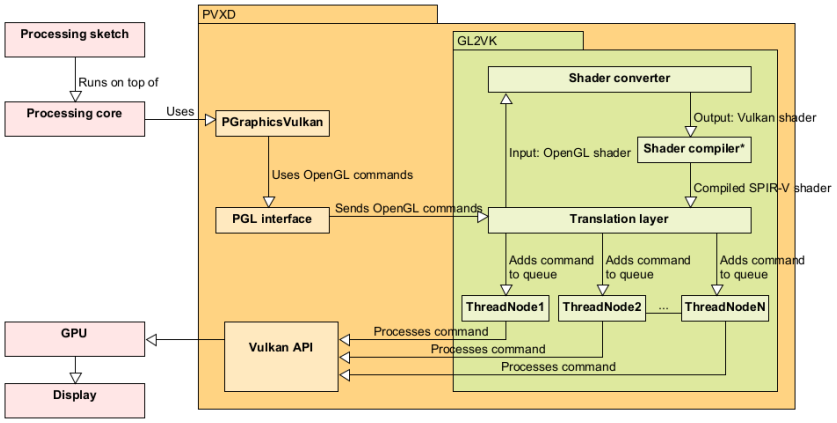


FIGURE 3.1: High-level graph view of the Vulkan renderer design

3.3.1 Translation layer

Processing's PXD code is thousands of lines in length; rebuilding the entire logic of the Processing renderer to use Vulkan would be a monumental task which could take too long to complete in the scope of this project. Fortunately, Processing's modular framework provides a relatively straightforward way to implement a Vulkan framework without rewriting the logic; by using the PGL abstraction layer. Calls through the PGL

layer are instead re-directed to an OpenGL-to-Vulkan translation layer. This layer is specifically optimised to perform well with Processing. Using a translation layer also means Processing's renderer does not need to be rewritten or even modified; the results from this logic are passed to PGL, which gets converted into Vulkan commands through the translation layer.

3.3.2 Shader converter

A core part of OpenGL and Vulkan is GPU shaders [Vul, 2025a]. Processing uses numerous shaders depending on what is drawn to the screen. To program shaders, they both use a language called GLSL to compile human-readable code into instructions for the GPU. However, Vulkan uses a different variant of GLSL and hence, Processing's OpenGL-based shaders will not be compatible with Vulkan. This component solves the problem by taking an OpenGL GLSL shader as an input and converting it into a Vulkan GLSL shader. This is done during runtime as Processing passes the shader code via the shaderSource() OpenGL command.

Another major difference is that Vulkan typically uses pre-compiled shaders (called "SPIR-V") which are compiled from source during development time, whereas OpenGL compiles the shaders from source during the program's runtime [Singh, 2016]. To solve this, PVXD contains a shader compiler library to compile GLSL shaders into SPIR-V during the runtime of the program, which is passed directly into Vulkan as a usable shader file.

3.3.3 Thread nodes

Vulkan opens the opportunity for multithreading. For this, the renderer uses a system similar to a master-slave configuration [Azevedo et al., 2011]; the main thread issues commands to external "worker" threads. These worker threads executes Vulkan commands, which typically have an overhead associated to it, and hence offload the main thread from this overhead. The main threads issue commands to thread-safe queues, and worker threads will pick and execute these commands. If a thread runs out of commands to execute, it sleeps until it receives a new command. Each worker thread has its own queue and Vulkan resources.

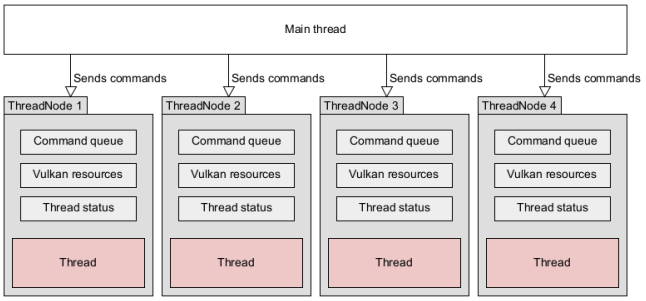


FIGURE 3.2: A high-level view of the thread node system; the main thread issues commands to thread via a queue.

3.4 Research Methodology

3.4.1 Evaluation plan

3.4.1.1 Specifications and details

Using a build of Processing with PVXD, 13 test sketches are benchmarked, using PXD and PVXD. Each sketch has code to record data and exit automatically once the required data has been collected. The tests are repeated 3 times to collect a mean average of the data. The following process is followed:

1. Run test sketch with PXD.
2. Wait until it automatically saves the data and exits.
3. Run test sketch with PVXD.
4. Wait until it automatically saves the data and exits.
5. Repeat steps 1-4 two more times.
6. Calculate mean average of the 3 runs.

The tests are performed on a Microsoft Surface Laptop Studio 2. The device specifications are listed as follows:

TABLE 3.3: Test device specifications

CPU	Intel Core i7-13700H
Base clock speed	2.90GHz
Maximum boost clock speed	5GHz
Cores	6 P-cores, 8 E-cores; 14 total
Logical processors	20
Memory	16GB LPDDR5x @ 2.60GHz
GPU	Intel Iris Xe Graphics
OS	Windows 11 Home, version 24H2, build 26100.3476

Additionally, the details for the Intel Iris Xe Graphics GPU are listed as follows:

Table 3.5: GPU specifications

Clock speed	1.5GHz
Execution units	96
Configuration	Integrated; UMA memory layout
Memory	8GB shared memory

3.4.1.2 Performance metrics

During the evaluation, the following variables is used:

Independent variable (graph x-axis): **Frame** - Elapsed frames since the sketch started.

Dependant variable (graph y-axis): **Interval** - The time between the current frame and the last frame, measured as:

$$Interval = (currentTimestamp - previousTimestamp)$$

Where timestamp is the system's current time (in nanoseconds) recorded at a specific point in the program. More specifically, below is a high-level representation of the test sketch's code. The interval data is represented in milliseconds to aid readability in the evaluation section.

```
long previousTimestamp = System.nanoTime();
while (programRunning) {
 // Execute some drawing code, for example, render many shapes.
 draw();
 // The draw commands are then dispatched to the GPU at this stage.
 long previousTimestamp = timestamp;
 // Update the timestamp.
 timestamp = System.nanoTime();
 currentTimestamp = timestamp;
 interval[frame] = (currentTimestamp - previousTimestamp);
 // Repeat until program closes.
}
```

3.4.1.3 Intel VTune

To meet objective O6, sketches are run through Intel VTune, separately from the benchmarking process. Intel VTune is a performance analysis tool which collects CPU, GPU, and memory usage information to identify bottlenecks, and show the hardware usage when programs are running [Int, 2025]. VTune collects this information by collecting hardware event-based samples, and calculates the metric based on the values between 2 sample points. The collected information is:

- **Logical Core CPU Time -** Defined from the VTune guide as "CPU Time is time during which the CPU is actively executing your application.".
- **EU Array Active -** Defined from the VTune guide as "The normalized sum of all cycles on all cores spent actively executing instructions.".
- **EU Array Idle -** Defined from the VTune guide as "The normalized sum of all cycles on all cores when no threads were scheduled on a core".
- **EU Array Stalled -** Defined from the VTune guide as "The normalized sum of all cycles on all cores spent stalled. At least one thread is loaded, but the core is stalled for some reason.", the guide states that this is typically caused by memory bandwidth bottlenecks.
- **EU Threads Dispatch -** Defined from the VTune guide as "The normalized sum of all cycles on all cores and thread slots when a slot has a thread scheduled.".
- **GPU Memory Access, GB/sec -** Defined from the VTune guide as "GPU memory read/write bandwidth between the GPU, chip uncore (LLC) and main memory. This metric counts all memory accesses that miss the internal GPU L3 cache or bypass it and are serviced either from uncore or main memory."
- **GPU Utilization -** Defined from the VTune guide as "The percentage of time when GPU engine was utilized."

Please note that Intel VTune are not part of the benchmarking tests, rather they exist to provide an insights and clues into the reason why the benchmarks perform the way they do; this is to gain an understanding of the results. This represents O7.

3.4.2 Benchmarking tests

Processing provides a set of performance-testing example sketches to test the speed of the Processing framework. These example sketches are used as benchmarking tests to ensure that they represent realistic sketches that a Processing programmer may create. These example sketches have been modified to record performance data. For tests that

use the random function, a pre-defined seed is set to ensure the same sequence of random numbers and guarentee that both OpenGL and Vulkan sketches are visually identical.

All tests and their details are listed below. Mode represents whether it runs in immediate, static retained, dynamic retained, or optimised retained rendering modes. Length represents the number of frames the test will run before it saves the data and closes. Screenshot references the figure that shows a screenshot of the test.

Table 3.7: Benchmarking tests

Name, Mode, Length, Screenshot	Description	Purpose
LineRendering, Immediate, 100, 5.1a	Thin transparent lines are drawn randomly on to the screen each frame, using immediate mode.	To test immediate mode geometry processing in an environment that is not limited by fillrate.
TextRendering, Immediate, 180, 5.1b	20000 small strings of text reading "HELLO WORLD" are rendered onto the screen in random positions, every frame. This uses GPU texturing.	To find out if there is any significant performance differences between OpenGL and Vulkan using basic texturing.
QuadRendering, Immediate, 100, 5.1c	Transparent squares are drawn on to the screen during this sketch's runtime. The number of squares initially starts at 10000. As the number of frames elapsed since the start of the sketch increase, the number of squares increases by 2000 per frame.	To test immediate mode geometry processing in an environment with large fillrates, and to observe scalability as the number of geometry increases.
CubicGridImmediate, Immediate, 360, 5.1d	A large rotating grid of 3D coloured transparent cubes are drawn to the screen using immediate mode. Depth buffering is deactivated so no rendering errors occur with transparency [Cohen-Or et al., 2000].	To test rendering of 3D geometry using immediate mode.

CubicGridRetained, Static retained, 360, 5.1d	Similar to CubicGridImmediate, but the cubic grid is drawn using retained mode. Geometry is created into a PShape object during startup, and the shape is drawn with a single shape() function call.	To test rendering of 3D geometry using retained mode.
CPUThrottleSingleCore, Optimised retained, 500, 5.1e	Small square shapes are drawn using retained mode, using a technique which allows each square to be rendered at a unique position at the cost of high CPU usage.	To find out if faster rendering can be achieved with the low-overhead Vulkan API compared to the OpenGL API.
CPUThrottleMultiCore, Optimised retained, 500, 5.1e	Similar to CPUThrottleMultiCore, except the multithreading feature in PVXD is used.	To compare with the CPUThrottleSingleCore test results and find out if Vulkan's multithreading support can improve performance over OpenGL, and by how much.
DynamicParticlesImmediate, Immediate, 1000, 5.1g	Renders physics-based textured objects which originate from a moving singular point. Immediate rendering is used.	To compare the performance difference of rendering moving objects in OpenGL and Vulkan using immediate mode.
DynamicParticlesRetained, Dynamic retained, 1000, 5.1g	Similar to DynamicParticlesImmediate, except using retained mode. A parent PShape object contains child objects which are created during startup, and each child object is moved using PShape's translate() function. Rendering all child objects is done by drawing the parent object once using shape().	To compare the performance difference of rendering moving objects in OpenGL and Vulkan using retained mode.
StaticParticlesImmediate, Immediate, 360, 5.1f	Fixed-position textured objects are rendered as the camera rotates. Immediate mode is used.	To measure the performance of rendering non-moving objects in OpenGL and Vulkan using immediate mode.

StaticParticlesRetained, Static retained, 360, 5.1f	Same as StaticParticles-Immediate, except retained mode is used. A parent PShape object contains child objects which are created during startup, and rendering all child objects is done by drawing the parent object once using shape().	To measure the performance of rendering non-moving objects in OpenGL and Vulkan using retained mode.
ManyTextures, Immediate, 1000, 5.1h	Uses same code as DynamicParticlesImmediate, except each object's texture is different from the last, resulting in texture switching.	To find out if using Vulkan can improve performance over OpenGL when excessive texture switching is performed.
GPUThrottle, Optimised retained, 200, 5.1i	Creates a PShape object with numerous vertices, and renders it 100 times per frame. The object is big and hence the fillrate is high in this sketch. This test is designed to maximise GPU usage so performance differences can be evaluated on the GPU.	To measure the difference between OpenGL and Vulkan purely on the GPU.

Chapter 4

Evaluation

4.1 Evaluation preface

4.1.1 Performance representation

The upcoming section will present each of the 13 tests, accompanied by 3 graphs; (a) is the line graph showing the time interval between each frame with OpenGL and Vulkan. (b) and (c), are Intel VTune graphs for OpenGL and Vulkan respectively. Note the VTune graphs should not be viewed as the final results for performance; their purpose is to provide an insight into potential causes for speedups, slowdowns, or bottlenecks. In other words, graph (a) aligns with objectives O4 and O5, whereas graphs (b) and (c) align with O6 and O7. As a guide for graph interpretation, the following conditions should hold true as an indicator of performance:

- Single-core CPU load, especially in the main thread, should be lower. Less single-core CPU usage means more work is being delegated to the high-performance GPU, and offloaded to worker threads in other CPU cores. If the CPU is carrying out geometry operations that the GPU should be handling, this will have a negative impact on performance, because the CPU performs these operations much slower than the GPU. Typically, this results in the GPU being in an idle state without any work to process.
- Multi-core CPU load (more than one core running at once) should be higher. Higher usage of other cores means work is being offloaded from the main thread, which gives it more time to complete important tasks.
- GPU load should be higher. More GPU usage means more geometry is being processed by execution units specifically optimised for this task, which is positive on performance.

- Memory access should be lower. Memory bandwidth can cause bottlenecks on both the CPU and GPU, which has a negative impact on performance.
- Stalling/idle rate on the GPU should be lower, and the active rate should be higher.

4.1.2 Full CPU and GPU usage

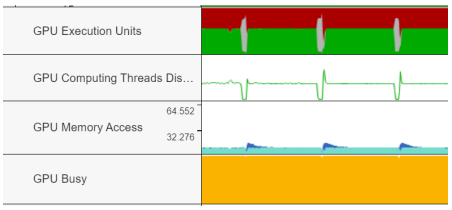
In other computing applications, utilising both CPU and GPU to full capacity is desirable, as this maximises computer capability. In this project however, it was just stated that less CPU usage results in better performance. At first glance, the idea of the CPU sharing some of the GPU's tasks is not illogical, especially if the GPU is at 100% usage. However, this is realistically not suitable for graphics programming; if the CPU were to share some of the GPU's tasks, it would eventually have to synchronise its resources with the GPU, which is likely to cause large slowdowns; this would result in worse performance than not sharing GPU tasks. Therefore, lower CPU usage, even if the GPU appears to be causing the bottleneck, is typical indicators of good performance.

4.1.3 Graph examples of good and bad performance

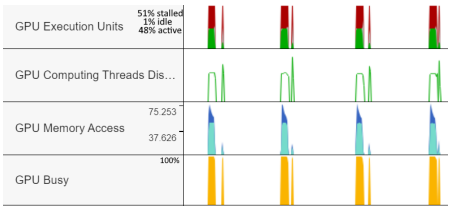
Additionally, figure 4.1 show ideal and non-ideal examples of CPU and GPU usage.

(A) An example of good CPU usage. Lower CPU time = better performance

(B) An example of bad CPU usage. Higher CPU time = worse performance.



(C) An example of good GPU usage. Higher GPU usage = better performance.

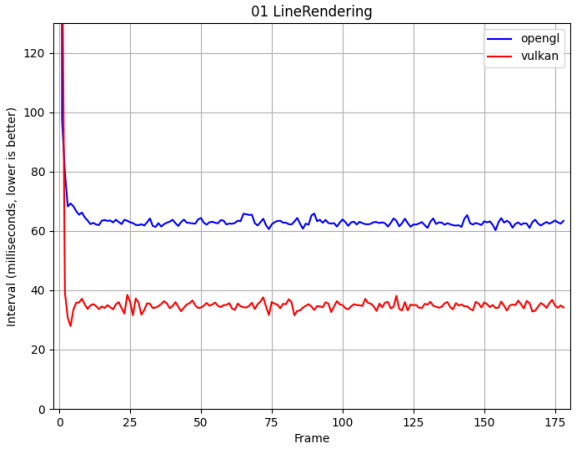


(D) An example of bad GPU usage. Lower GPU usage = worse performance.

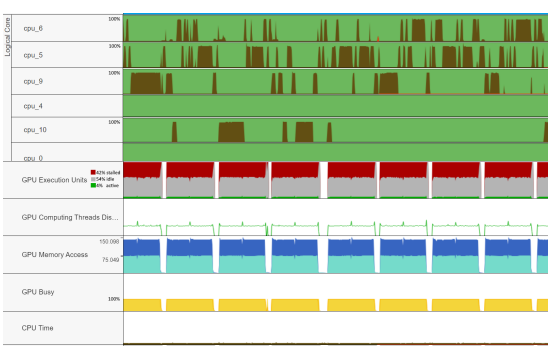
FIGURE 4.1: VTune graphs showing good and bad examples of CPU and GPU usage.

4.2 Evaluation results

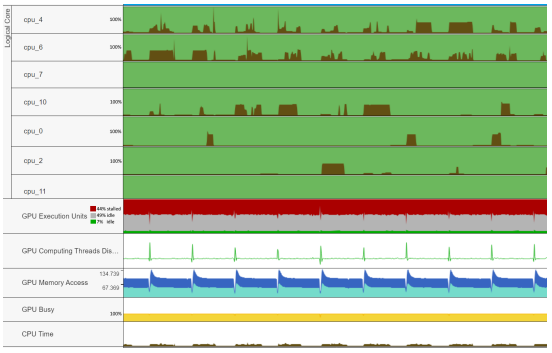
Test 01 - LineRendering



(A) LineRendering test results



(B) OpenGL VTune view, zoomed in



(C) Vulkan VTune view, zoomed in

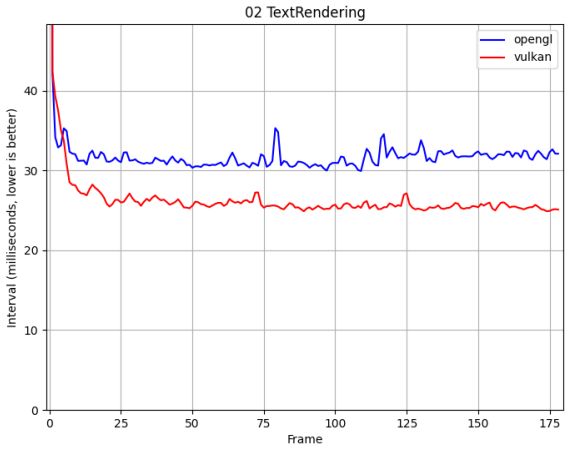
FIGURE 4.2: LineRendering test results and VTune graphs

In this test, there appears to be a large difference in interval times between Vulkan and OpenGL; Vulkan is roughly twice as fast as OpenGL. The VTune graphs shows noticable reasons for this improvement; the use of multithreading means the Vulkan renderer can offload expensive buffering operations to other cores; we can see that OpenGL, by comparison, only ever runs on one core at a time.

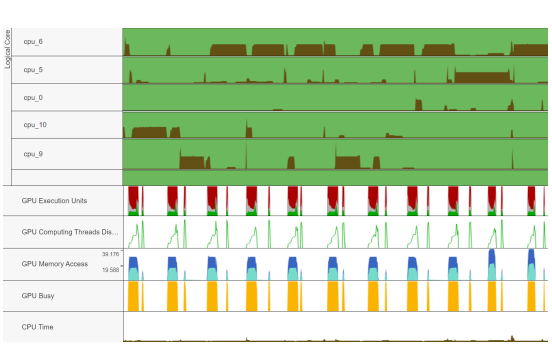
This is further backed up by the GPU's busy time; in the OpenGL VTune graphs, we can see the GPU running out of tasks and pausing all activity. This is most likely because the CPU is acting as the bottleneck and is not issuing commands quickly enough to the GPU. By comparison, the Vulkan VTune results show that the GPU is constantly active; this indicates that the CPU is providing a constant stream of commands so the GPU never runs out of tasks, hence avoiding the limitation in OpenGL. Another contributing

factor memory access; the memory access in Vulkan (approx. 45GB/s read, 43GB/s write) is about 39% lower than OpenGL's access rate (approx. 71GB/s read, 67GB/s write).

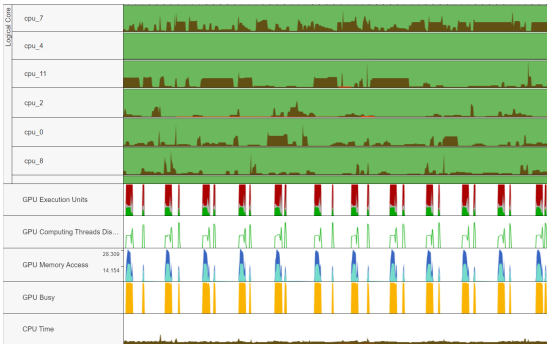
Test 02 - TextRendering



(A) TextRendering test results



(B) OpenGL VTune view, zoomed in



(C) Vulkan VTune view, zoomed in

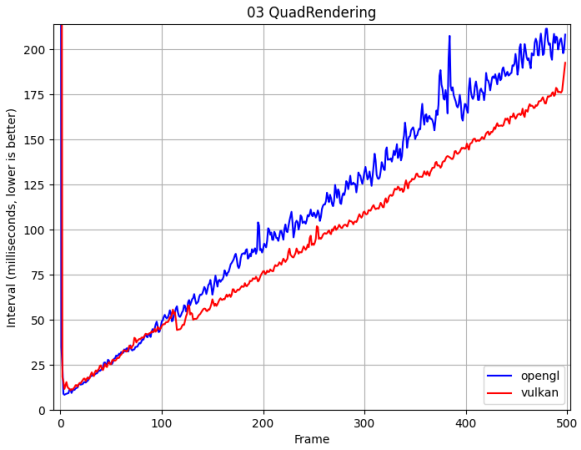
FIGURE 4.3: TextRendering test results and VTune graphs

4.3a shows a interval during the runtime of the sketches on both OpenGL and Vulkan, with only a small performance improvement on Vulkan. The VTune graphs show that the GPU is underutilised, completing its tasks much faster than the CPU, making the CPU the bottleneck in this test. Both OpenGL and Vulkan show an extremely high active time in the main thread, which indicates that performance bottlenecks reside in the handling of text and geometry.

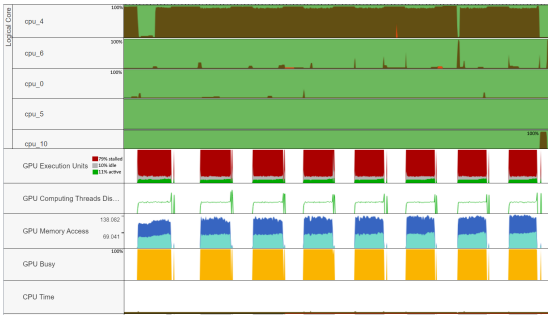
There is no evidence to suggest that the use of texturing (for the text) causes this high CPU usage. The high CPU usage could be caused by the logic handling generation

of visible text in the sketch, which would not be related to the overhead of OpenGL or Vulkan. The results also indicate that the slight performance boost arises from the Vulkan renderer is offloading buffering operations to separate CPU cores.

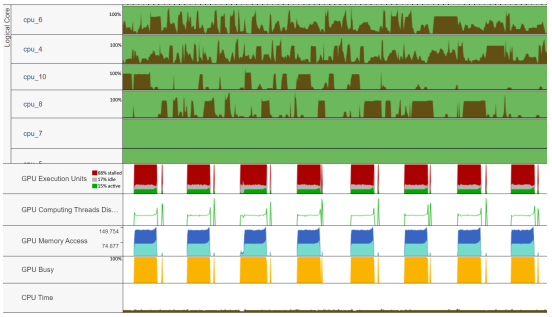
Test 03 - QuadRendering



(A) QuadRendering test results



(B) OpenGL VTune view, zoomed in



(C) Vulkan VTune view, zoomed in

FIGURE 4.4: QuadRendering test results and VTune graphs

As the frames elapsed increases in the sketch, the number of geometry rendered per frame increases. Hence, we can observe a linear shape to the graph as the interval increases during the runtime of the sketch, both on OpenGL and Vulkan. However, Vulkan shows an interval drop of 8.3 milliseconds on frame 110; figure 4.5 show a sudden memory access decrease at this point in the program (from 60GB/sec read to 46GB/sec read), although there is no clear explanation of this sudden memory decrease.

The sketch is CPU bottlenecked throughout its runtime on both OpenGL and Vulkan, and this only worsens as the number of geometry increases. This indicates that the

GPU is able to complete its tasks extremely quickly, despite the high fillrate of rendering quadrilaterals in the sketch.

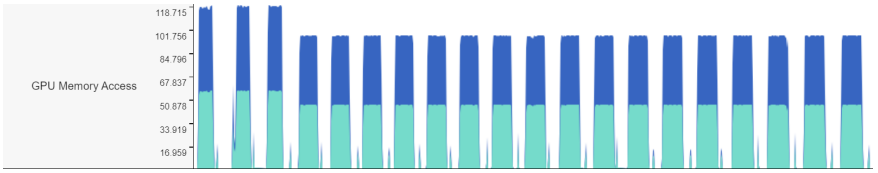
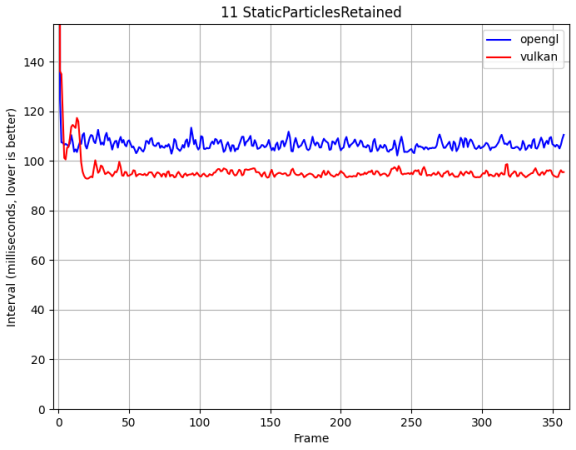
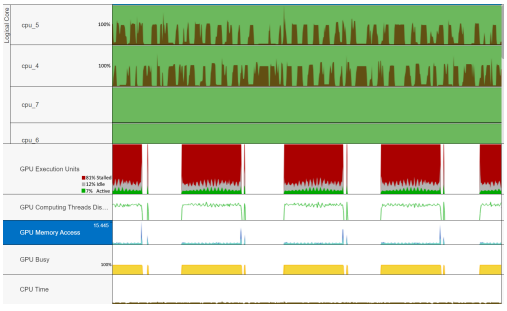


FIGURE 4.5: GPU memory access showing sudden decrease in memory access.

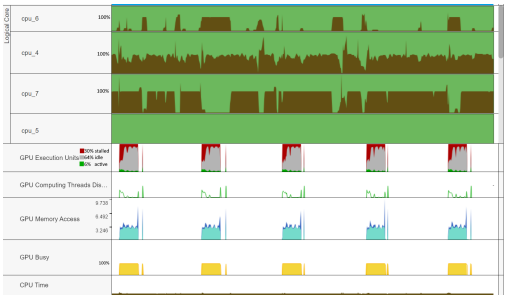
Test 04 - CubicGridImmediate



(A) CubicGridImmediate test results



(B) OpenGL VTune view, zoomed in



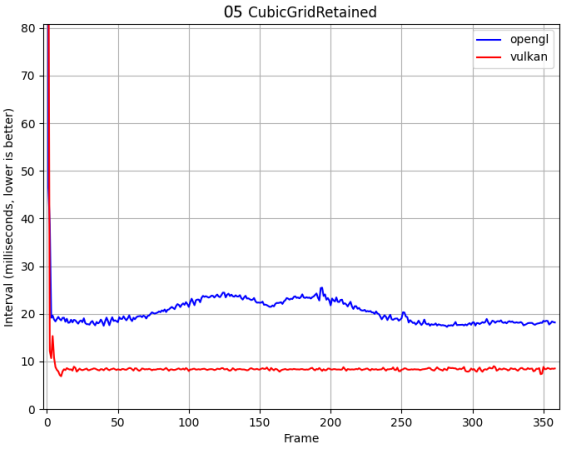
(C) Vulkan VTune view, zoomed in

FIGURE 4.6: CubicGridImmediate test results and VTune graphs

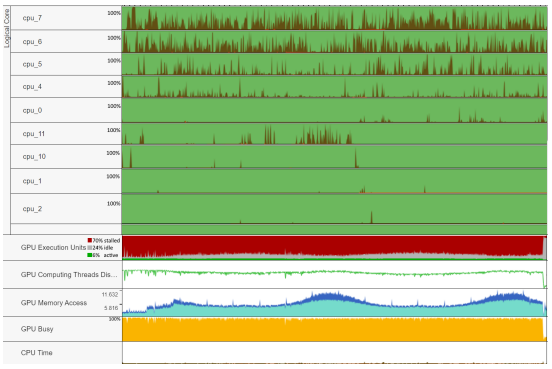
Figure 4.6a shows a relatively constant latency on both OpenGL and Vulkan renderers with Vulkan only having a slight improvement over OpenGL by around 10 milliseconds. The VTune graph results indicates that the program is CPU-bottlenecked; this is apparent if we check the sketch's code. The sketch makes use of pushMatrix(), translate(), and popMatrix(), which causes matrix operations to be performed on the CPU per vertex, possibly largely contributing to the bottleneck.

The VTune graphs also reveals something interesting on the GPU side: OpenGL has a much higher stall rate than Vulkan, and the memory access speed is lower than Vulkan's access speed. The high stall rate could correspond to the lower memory access rate, although the cause of this low memory bandwidth is unknown.

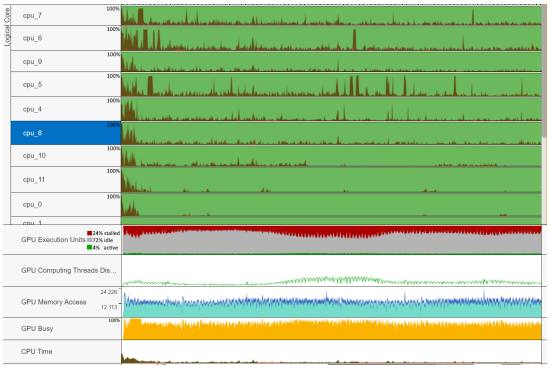
Test 05 - CubicGridRetained



(A) CubicGridRetained test results



(B) OpenGL VTune view, zoomed out



(C) Vulkan VTune view, zoomed out

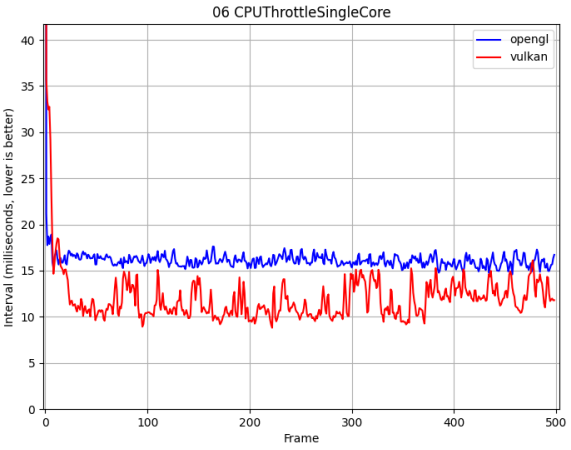
FIGURE 4.7: CubicGridRetained test results and VTune graphs

OpenGL appears to have waves of increased interval times. This could be explained by the rotation of the geometry in the sketch; as the cubes get closer to the camera, the fillrate increases, increasing GPU usage, and hence increasing the latency between frames. Vulkan does not appear to be affected by this, and maintains a mostly constant interval thoughout its runtime.

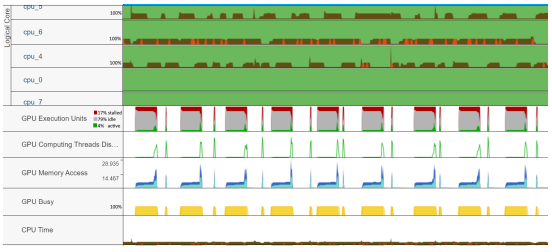
VTune reveals several interesting observations about this test:

- The CPU time on OpenGL is much higher than Vulkan; even taking into consideration the fact that the Vulkan renderer offloading operations to other CPU cores, this is still unusually high for OpenGL. This could be explained by the overhead of OpenGL function calls.
- Similar to 4.6, the GPU execution units stall time is much higher on OpenGL than Vulkan, and the memory access is much lower on OpenGL than Vulkan; this could suggest a memory bandwidth bottleneck, though it is not clear what this is caused by.
- The fillrate in Vulkan is very high; this suggests that OpenGL suffers from a poor fillrate due to accessing memory poorly.

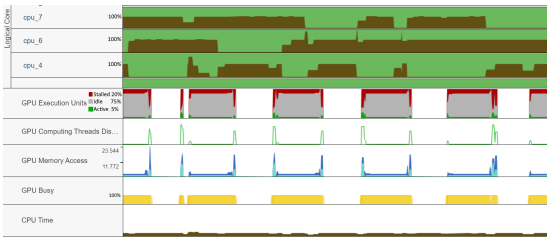
Test 06 - CPUThrottleSingleCore



(A) CPUThrottleSingleCore test results



(B) OpenGL VTune view, zoomed out



(C) Vulkan VTune view, zoomed out

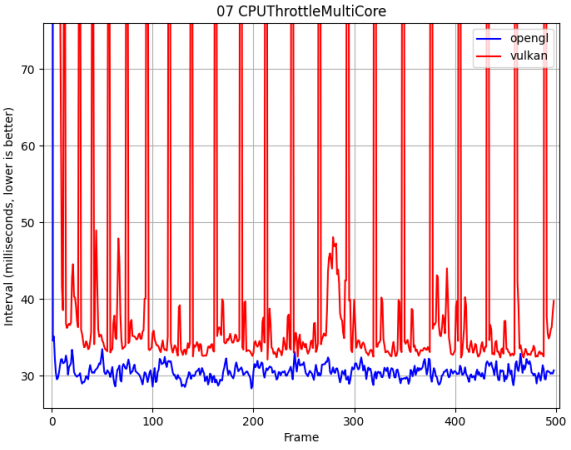
FIGURE 4.8: CPUThrottleSingleCore test results and VTune graphs

OpenGL shows mostly constant intervals throughout its runtime, whereas Vulkan shows more varied results, i.e. the interval fluctuates randomly and frequently. Vulkan performs slightly better than OpenGL with an approximate 7 milliseconds improvement.

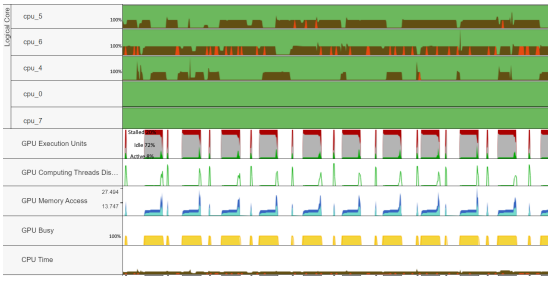
The VTune graphs show that excessive use of OpenGL functions causes spinning in the CPU as shown by the red markers areas in the CPU view, whereas Vulkan doesn't appear to suffer from such stalls. Interestingly, the OpenGL VTune graphs show a separate

thread working concurrently to the main thread; this could indicate the OpenGL driver is offloading tasks into another thread, but there is no other evidence to back this up, so this thread's functionality is unknown.

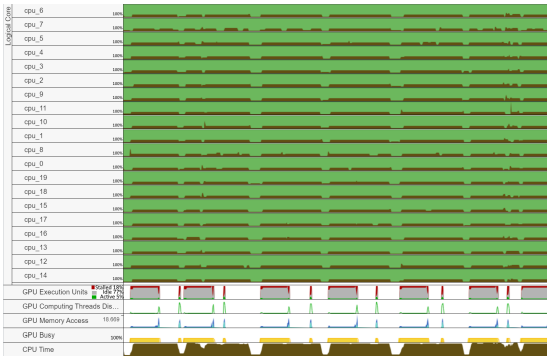
Test 07 - CPUThrottleMultiCore



(A) CPUThrottleMultiCore test results



(B) OpenGL VTune view, zoomed out



(C) Vulkan VTune view, zoomed out

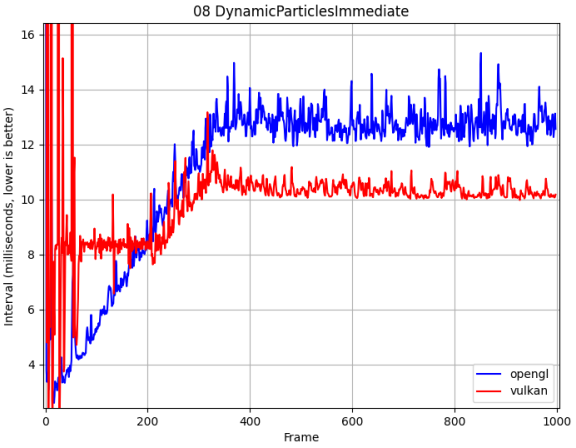
FIGURE 4.9: CPUThrottleMultiCore test results and VTune graphs

There is 2 standout behaviours shown in the graph: Vulkan is slower than OpenGL, and Vulkan exhibits large slowdown spikes at routine intervals. This is a typical indicator of excessive garbage collection and indicates that there is a flaw in the Vulkan renderer that is generating too much garbage per command call. However, these spikes should be ignored; if the spikes weren't there, the Vulkan renderer would most likely continue to perform worse than OpenGL.

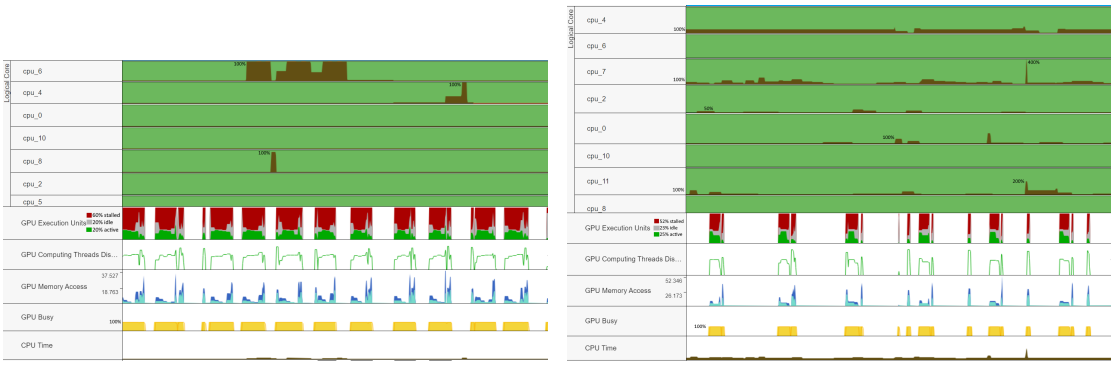
4.9b and 4.9c show that there is indeed parallelism being used in Vulkan, whereas

OpenGL with its lack of multithreading support, has similar behaviour to 4.8b. However, this parallelism appears to be degrading the performance rather than improving it on Vulkan. This is most likely caused by the overhead of thread scheduling in the operating system, and the Vulkan renderer's multithreading system appears to be flawed when the core count increases.

Test 08 - DynamicParticlesImmediate



(A) DynamicParticlesRetained test results



(B) OpenGL VTune view, zoomed out

(C) Vulkan VTune view, zoomed out

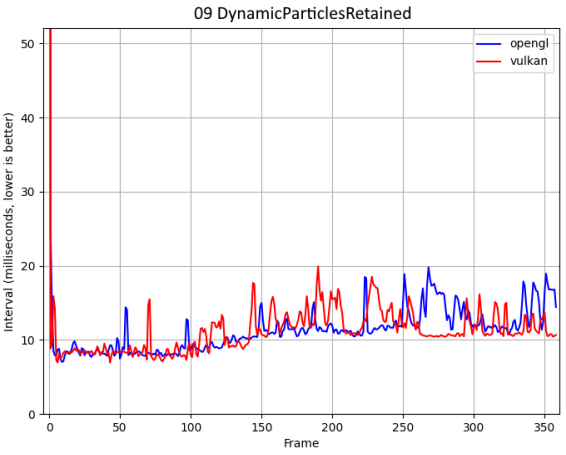
FIGURE 4.10: DynamicParticlesImmediate test results and VTune graphs

In 4.10a, OpenGL linearly increases from 0 milliseconds to around 13 milliseconds as more and more graphical elements are generated, before reaching the maximum allowed elements limit. Vulkan also exhibits this behaviour but only between frames 206 and 318; the beginning of the runtime in Vulkan shows large interval spikes followed by noisy but essentially constant intervals before frame 206. These large spikes could be explained by the Vulkan renderer's memory allocation process as it calls memory allocation functions

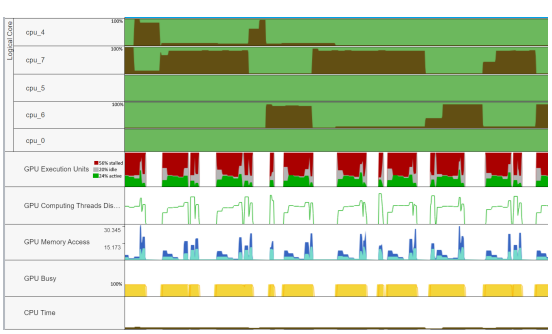
due to increased bufferData() calls. After frame 200, OpenGL's interval time exceeds Vulkan's interval time, which means that Vulkan is faster than OpenGL beyond this point.

The VTune graphs, 4.10b and 4.10c, show that the GPU completes its tasks a lot faster on Vulkan compared to OpenGL, and that the CPU acts as the bottleneck on both OpenGL and Vulkan, although this is partially alleviate on Vulkan though offloading buffering commands to separate threads, as shown in the CPU activity in 4.10c.

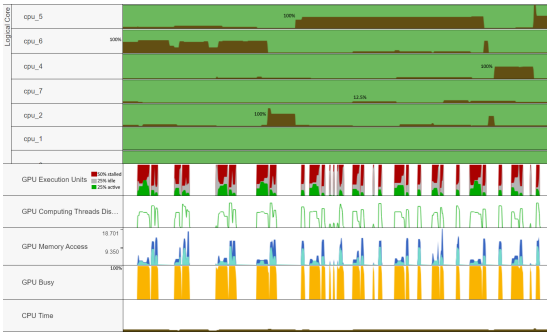
Test 09 - DynamicParticlesRetained



(A) DynamicParticlesRetained test results



(B) OpenGL VTune view, zoomed out



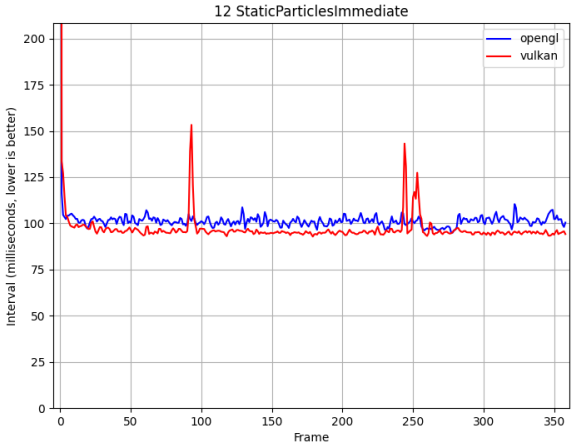
(C) Vulkan VTune view, zoomed out

FIGURE 4.11: DynamicParticlesRetained test results and VTune graphs

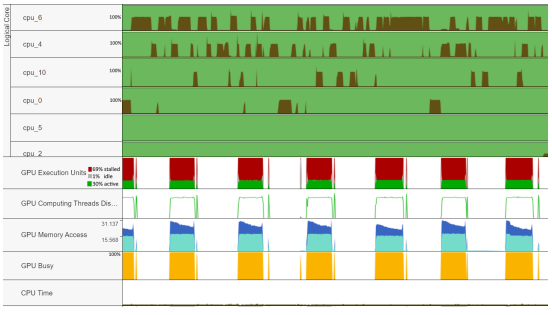
In this test, the results from 4.11 show that there is no clear victor in terms of performance; both OpenGL and Vulkan have the same general shape on the graph. This is significant, because it shows something interesting about the Processing framework, especially when using dynamic retained rendering.

As mentioned in Chapter 3, dynamic retained mode performs similarly to immediate mode in that it re-calculates and buffers vertex data every frame. There is however, one major difference, as shown by this test: the buffering process takes place outside of the GL2VK code, therefore buffering operations are not offloaded to worker threads in the Vulkan renderer. This is the reason the performance doesn't appear to improve in this test; this test is CPU-bound, and hence the interval cannot be improved by using Vulkan in dynamic retained mode.

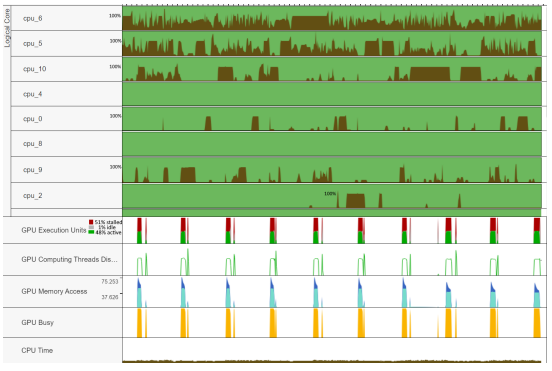
Test 10 - StaticParticlesImmediate



(A) StaticParticlesImmediate test results



(B) OpenGL VTune view, zoomed out



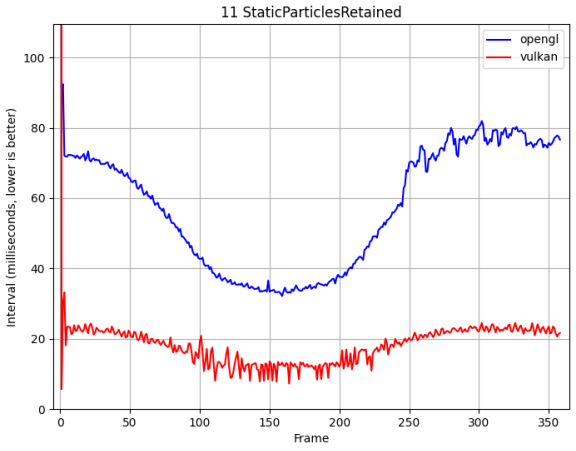
(C) Vulkan VTune view, zoomed out

FIGURE 4.12: StaticParticlesImmediate test results and VTune graphs

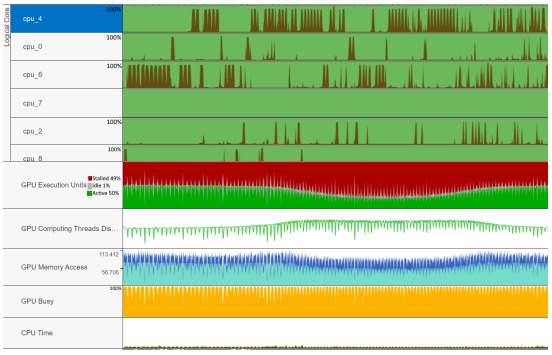
Similar to CubicGridImmediate, the line graph shows a relatively constant latency on both OpenGL and Vulkan with Vulkan only having a slight improvement. The VTune graphs, 4.12b and 4.12c reveal that both OpenGL and Vulkan are heavy on CPU usage, with Vulkan offloading buffering operations to separate cores.

Analysis of the test code shows that much like CubicGridImmediate, the StaticParticles-Immediate performs a large amount of operations per element; 4 vertices are generated via the vertex() function, and state-changing, performance-heavy functions beginShape() and endShape() are used per element rendered. While this is not direct evidence, it is a possible explaination for this high single-core CPU usage.

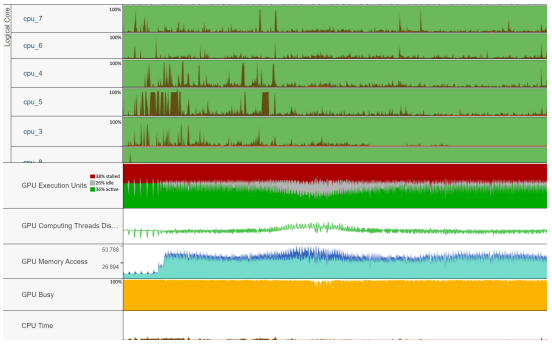
Test 11 - StaticParticlesRetained



(A) StaticParticlesRetained test results



(B) OpenGL VTune view, zoomed out



(C) Vulkan VTune view, zoomed out

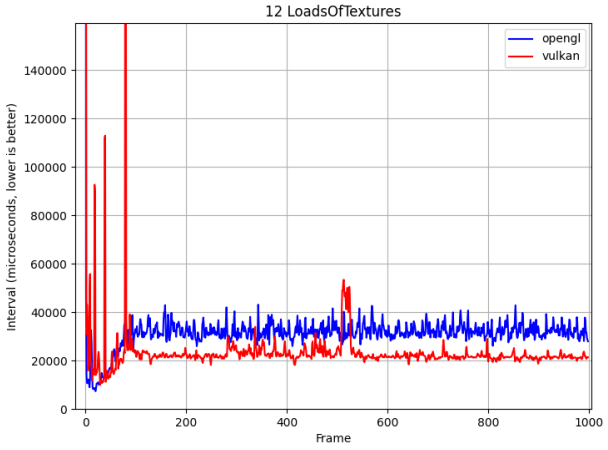
FIGURE 4.13: StaticParticlesRetained test results and VTune graphs

This graph shows a large increase in performance on Vulkan; it is approximately 3.5 times faster than OpenGL. On the OpenGL line, we can observe that as the graphical elements rotate, an inverse bell curve shape is formed on the graph, most likely due to certain viewpoints of the scene resulting in less elements being drawn on-screen. This is true for Vulkan too, but to a lesser extent.

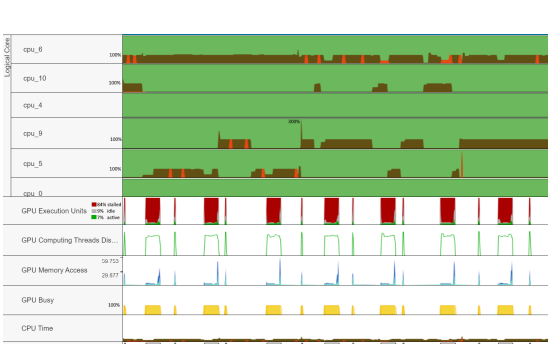
This performance increase is largely attributed to the GPU; the VTune graphs show that:

- The CPU usage does not pose any performance bottlenecks, even though OpenGL's CPU usage is high.
- In Vulkan, the GPU stall rate is lower and the active rate is higher. Additionally, the memory access is significantly lower in Vulkan (read: 34GB/s, write: 1GB/s) compared to OpenGL (read: 69GB/s, write: 21GB/s).

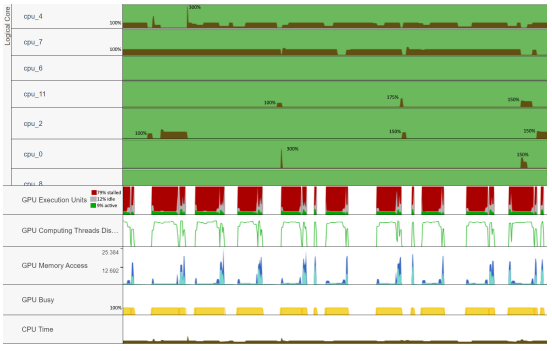
Test 12 - ManyTextures



(A) GPUThrottle test results



(B) OpenGL VTune view, zoomed out



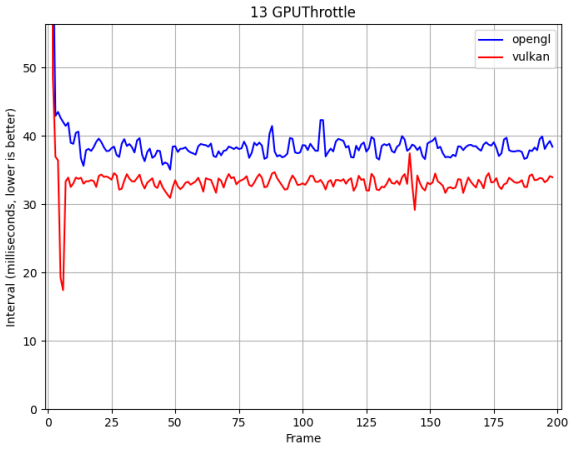
(C) Vulkan VTune view, zoomed out

FIGURE 4.14: ManyTextures test results and VTune graphs

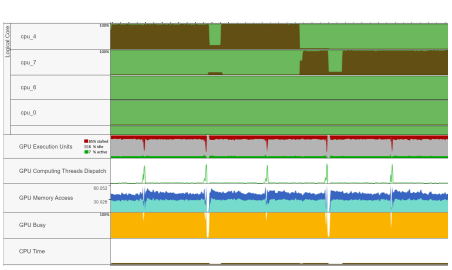
Vulkan appears to show spikes of large intervals during its runtime; this could be caused by memory allocation calls due to frequent bufferData() calls. What VTune reveals is that there is CPU stalling on OpenGL; this test makes use of an OpenGL function called bindTexture(), which may be causing the slowdown as this function is not used in previous tests. Vulkan does not appear to be affected by binding textures on the

CPU side. However, it is worth noting that both OpenGL and Vulkan renderers have very high stall times on the GPU Execution Units row. This suggests that frequently changing texture is not ideal for the GPU hardware in terms of performance, regardless of what API is used.

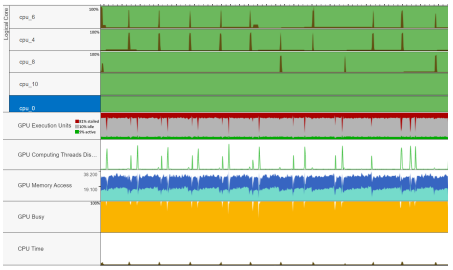
Test 13 - GPUThrottle



(A) GPUThrottle test results



(B) OpenGL VTune view, zoomed out



(C) Vulkan VTune view, zoomed out

FIGURE 4.15: GPUThrottle test results and VTune graphs

This test shows only a small performance improvement compared to other GPU-bound benchmarks like ManyTextures or CubicGridRetained. This may indicate that the GPU is better suited to pixel processing rather than vertex processing.

Though not related to the test, 4.15b and 4.15c show that Vulkan CPU usage is much lower in Vulkan than OpenGL. This could suggest a high overhead of calling OpenGL functions. In an unexpected twist, CPUThrottleSingleCore and CPUThrottleMultiCore did not show significant speedup due to Vulkan's reduced overhead. Here however, there is evidence that performance can be significantly improved through CPU overhead.

4.3 Discussion

From all the tests above, we can discern the following:

- On Vulkan, performance is increases up to 3.5 times compared to OpenGL, assuming that dynamic retained rendering is not used.
- Depending on the implementation of the sketch, immediate mode rendering with Vulkan can be up to 2 times faster than OpenGL. However, this varies significantly, and in most cases, it is most likely to only be a 20%-50% performance increase.
- Large CPU load on the main thread is the most likely cause for bottlenecks, depending on the implementation of the sketch. This is due to operations such as vertex calculations, tesselation, buffering, or miscellaneous logic on the Processing core, outside of the GL2VK code.
- The graphs from VTune reveal that memory bottlenecks contribute to high stalling and idle rates, and that OpenGL has a higher memory footprint on the GPU compared to Vulkan.
- While multithreading aids in offloading heavy operations such as buffering, using too much CPU cores can cause slowdowns instead of improving performance.
- Texture switching causes high stall rates in the GPU, on both OpenGL and Vulkan.
- On Vulkan, there are no performance improvements when using dynamic retained rendering. This is because the CPU acts as the bottleneck, and all buffering operations (that would normally be offloaded to separate CPU cores) are performed on the Processing core in the main thread, outside of the Vulkan renderer.

Chapter 5

Conclusion

5.1 Summary

The tests show that in general, better performance can be achieved by using the Vulkan renderer instead of OpenGL, and performance improvements highly varies depending on the sketch. Basic sketches using immediate mode can perform between 20%-100% faster than OpenGL. More advanced sketches can achieve up to 3.5 times the performance of OpenGL, provided the programmer is aware of the techniques to achieve this performance.

The tests also show the limitation of this current implementation with Vulkan; numerous tests were bottlenecked by the CPU. This is mostly due to processes such as tesselation, buffering, and miscellaneous logic. The Vulkan renderer was able to offload these buffering operations to other cores which improved performance, but the rest of the operations such as tesselation was delegated to the main thread; this was outside the Vulkan renderer's control. In theory, these operations could also be offloaded from the main thread and to other CPU cores; this should improve performance considerably.

There are also areas in the Processing renderer which are specifically optimised for better performance with OpenGL, but do not improve performance with the Vulkan renderer. A major example of this is DynamicParticlesRetained. Much of the geometry is calculated on the CPU to avoid expensive OpenGL API calls. This could be redesigned to take advantage of Vulkan's faster, multithreaded API calls, and offload geometry rendering

to the GPU. It could also be redesigned to take advantage of Vulkan-specific features, such as descriptor sets.

5.2 Limitations

While this project demonstrated what Processing would look like if it were to use Vulkan, it is important to keep in mind that it does not represent a perfect Vulkan renderer. The renderer used the PGL abstraction layer to translate OpenGL command to Vulkan commands, which massively simplifies the implementation process, as the OpenGL renderer contains tens of thousands of lines of code; refactoring this would have been far outside of the scope of this project, and would have taken a long time to complete. As such, there are several limitations with the current implementation of the Vulkan renderer:

5.2.1 Implementation limitations

- *Unfit for Vulkan:* Processing was designed with OpenGL in mind. Hence, best practices for OpenGL optimisation are used in the Processing framework. The problem is that most of these optimisations do not carry over to Vulkan, and in some cases harm performance rather than make it faster.
- *Retained mode:* Retained mode in Processing is not ideal for a Vulkan renderer; it is very similar to then immediate rendering mode in which it re-calculates moved vertices on the CPU. This is a task that could be significantly sped up if it were offloaded to the GPU instead.
- *CPU bottlenecks:* The tests have shown that significant CPU time is used in the tesselation and generation of geometry. These could theoretically be offloaded to separate CPU cores much like offloading buffering operations, but it could be argued that doing so would be outside of the scope of the project: The tesselation and generation code technically doesn't relate to any OpenGL or Vulkan code.
- *Multithreading:* Multithreading in the Vulkan implementation is ultimately flawed as it causes slowdowns rather than speedups if too much cores are assigned. This may be caused by the overhead of multithreading outweighing any performance benefits, and could theoretically be solved with more efficient scheduling and command issuing code.

5.2.2 Evaluation limitations

The results from the evaluation opens some further questions which do not have enough evidence for an explanation in this project. Additionally, Intel VTune could not gather

full debugging information due to the fact that a Java program using bytecode was being benchmarked. This severely limited the ability to gain more insight into the inner workings of OpenGL and Vulkan. Had it worked, it would have been significant for gaining evidence to further explain the behaviours observed. As such, the evaluation in this project suffers from these main limitations:

- There is no evidence to explain the GPU memory access differences between the Vulkan and OpenGL frameworks.
- There is no evidence to explain the reason for the high stall rates in all the tests.
- There is a lot of ambiguity in the inner workings of the OpenGL and Vulkan drivers, including why OpenGL was using 2 threads in CPUThrottleSingleCore.
- Benchmarking individual parts of the Processing code - which could have provided more explanations about the inner workings of the tests - were not performed and was considered out of scope for this project.
- In LineRendering, Vulkan has an unexpectedly high difference in interval time compared to OpenGL, which contradicts some of the results in other tests; Vulkan has shown to excel at high fillrates, which is not the case for this test. Vulkan has shown to only provide a slight performance increase in immediate mode, whereas this test provides a large performance increase in this test. There is no definite explanation as to what causes the large performance increase.

5.3 Future of Processing

As of the writing of this paper, there are ongoing efforts to modernise Processing, while ensuring with older sketches work correctly in newer versions. Part of this endeavour involves removing deprecated libraries. One such library is JOGL, the interfacing library for OpenGL in Java. Fittingly, this project replaces an increasingly obsolete API with a newer one; as discussed in Chapter 2, one of the problems identified was the eventual obsolescence of OpenGL.

The Processing Community lead Raphaël de Courville, and Processing contributor Stef Tervelde invited the author of this paper to a meeting to discuss this project. What was made immediately apparent is that backwards compatibility with existing sketches was the top priority, whereas this project focuses on speed.

This meeting provided an interesting insight into the compatibility issue. Currently, the version of the Vulkan renderer only works with a very specific subset of Processing

sketches, while most other sketches do not display correctly or crash. Theoretically, full backwards compatibility could be achieved while taking advantage of the speed of the Vulkan framework, but this would take significant development time.

An approach that the Processing developers are planning to explore is the use of Google's ANGLE [Goo, 2025]. ANGLE was briefly discussed in Chapter 2 as a translation layer that translates WebGL into Direct3D 9 in Google Chrome. However, modern releases of ANGLE outside of Google Chrome can translate OpenGL ES to other graphics APIs, including Vulkan. Using such a framework would take much less development time than using Vulkan natively. This would solve the JOGL obsolescence problem and future-proof Processing should support for OpenGL cease on certain platforms.

However, this could be a concern for performance, especially in the future. Using ANGLE would mean that only OpenGL ES features can be used; features specific to Vulkan (some of which can aid performance) cannot be used. For example, Vulkan supports multithreading; several tests in Vulkan had an advantage over the OpenGL renderer due to offloading expensive operations to separate cores, which cannot be done in OpenGL. Kligge has found that a Vulkan-like API, WebGPU, performs better than the OpenGL-based WebGL, which uses Google's ANGLE [Kligge, 2024]. This suggests that natively using Vulkan rather than through ANGLE is faster.

5.4 Future work

The results shown in this paper has proven that Processing can benefit from utilising Vulkan. However, this is only one element that was discovered during this research; bottlenecks were discovered that were not caused by OpenGL, and the meeting with the Processing developers has brought forward the issue of backwards compatibility. From this, several recommendations can be formulated.

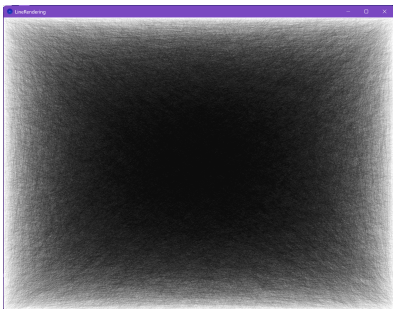
- **Offloading tesselation and geometry processing to other threads -** Most of the bottlenecks in the tests were caused by the tesselation and geometry generation logic; it may be possible to offloaded these operations into worker threads, which would coincide with Vulkan's multithreading support, and improve performance considerably.
- **Redesign retained mode -** The DynamicParticlesRetained showed that retained mode was heavily limited by the fact that geometry calculations were being

calculated on the CPU, which is not ideal for performance; using Vulkan provides a strong opportunity to better optimise retained mode and move geometry calculations to the GPU, which would be much more efficient.

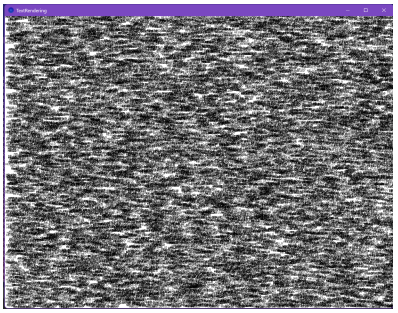
- **Testing on other hardware -** The tests were performed on only one device, which means that the data is limited for a framework that runs on a range of different devices. The tests could be repeated on several different systems with different GPUs. It is also worth noting that the tests were performed on integrated graphics with a UMA configuration; further testing could be completed on discrete graphics cards where the results are likely to be different.
- **Experimentation with Google's ANGLE -** The renderer used in this paper does not have good compatibility with some Processing sketches, so an alternative solution could be to use Google's ANGLE. Using ANGLE would be much faster in development time by comparison. This may not be the most efficient solution in terms of performance, but it may be possible to gain native access to Vulkan by modifying ANGLE's source code. If this is achieveable, this means that both OpenGL and Vulkan can be used at the same time; Processing would benefit from backwards compatibility (where needed) and faster performance.

Additionally, using Vulkan in Processing opens interesting opportunities, which could vastly benefit Processing in areas other than performance:

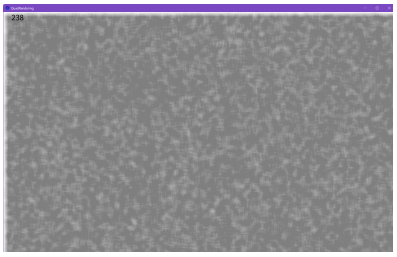
- **Vulkan abstraction layer for learning Vulkan -** Similar to PGL, a Vulkan abstraction layer would allow interfacing with Vulkan in the Processing environment. There are programmers who have learned OpenGL through Processing, because of its beginner-friendly coding interface which hides complicated boilerplate. Vulkan is considerably difficult to learn, and requires a lot of boilerplate code before anything can be rendered. Introducing access to native Vulkan within Processing sketches could massively improve the process of learning Vulkan.
- **More options for using Vulkan -** If a programmer wants to use OpenGL, there is a wide range of engines, libraries, tools, and frameworks that can aid with development using OpenGL. With Vulkan, its only major significance is in high-end game engines such as Unity and Unreal Engine, which do not expose the specific API interactions, and can hinder flexibilty to the programmer [Campbell, 2023]. The only other option would be to write code from scratch using Vulkan, which is a difficult process. Introducing Vulkan to Processing provides a new option for using Vulkan, which could bridge the gap between low-level coding from scratch, and high-level game engines.
- **Usability studies -** A study involving participants could be carried out to discover how much easier it is to learn and use Vulkan in Processing, compared to learning and using it from scratch.



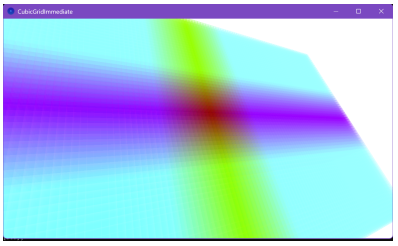
(A) LineRendering



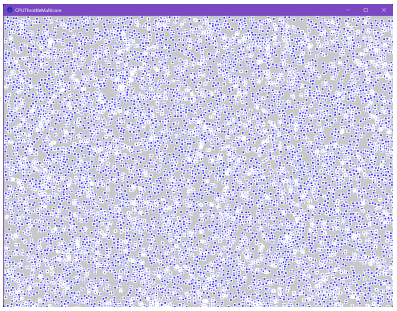
(B) TextRendering



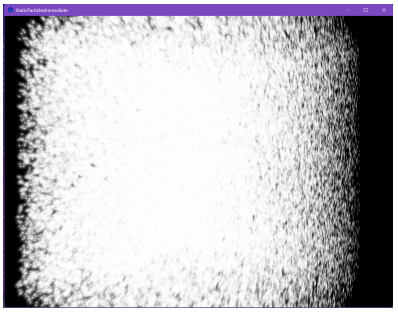
(C) QuadRendering



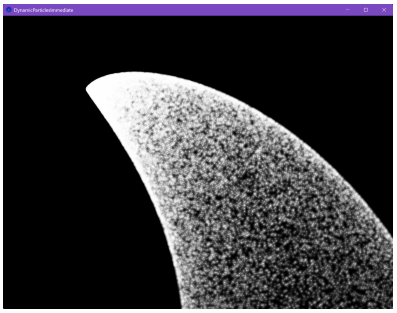
(D) CubicGridImmediate/CubicGridRetained



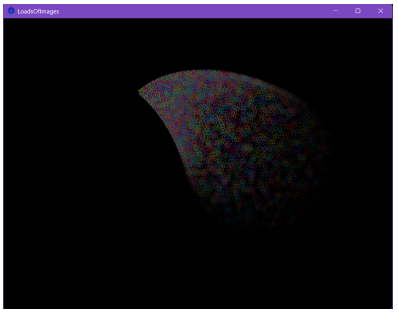
(E) CPUThrottleSingleCore/CPUThrottleMultiCore



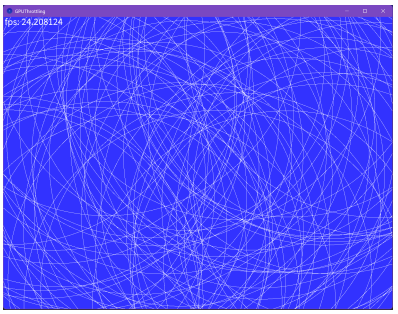
(F) StaticParticlesImmediate/StaticParticlesRetained



(G) DynamicParticlesImmediate/DynamicParticlesRetained



(H) ManyTextures



(I) GPUThrottle

FIGURE 5.1: Screenshots of the test sketches.

Bibliography

(2025). Google angle github. Available at https://github.com/google/angle.

(2025). Intel vtune user guide. Available at https://cdrdv2.intel.com/v1/dl/getContent/773628?fileName=vtune-profiler_user-guide_2023.1-766319-773628.pdf.

(2025). Openframeworks website - introducing opengl for of. Available at https://openframeworks.cc/ofBook/chapters/openGL.html.

(2025). Opengl website. Available at https://www.opengl.org.

(2025). Processing website. Available at https://processing.org.

(2025a). Vulkan shaders. Available at https://docs.vulkan.org/spec/latest/chapters/shaders.html.

(2025b). Vulkan website. Available at https://www.vulkan.org.

Allison, C. J., Zhou, H., Munawar, A., Kazanzides, P., and Barragan, J. A. (2024). Fire-3dv: Framework-independent rendering engine for 3d graphics using vulkan. In *2024 Eighth IEEE International Conference on Robotic Computing (IRC)*, pages 104–111. IEEE.

Amador Cambronero, N. et al. (2021). Digbuild: Sandbox voxel game and engine using vulkan.

Azevedo, G. M., Bradaschia, F., Cavalcanti, M. C., Neves, F. A., Rocabert, J., and Rodriguez, P. (2011). Safe transient operation of microgrids based on master-slave configuration. In *2011 IEEE Energy Conversion Congress and Exposition*, pages 2191–2195. IEEE.

Bodurri, K. F. (2019). Comparative performance analysis of vulkan and cuda programming model implementations for gpus. B.S. thesis.

Campbell, D. (2023). *Design of a High-performance Computer Graphics Interface in a High-level Programming Language*. PhD thesis, Wichita State University.

Chandra, D. (2018). *Developing a Simulation Framework for Vulkan*. North Carolina State University.

Chibalashvili, A., Savchuk, I., Olianina, S., Shalinskyi, I., and Korenyuk, Y. (2023). Creative coding as a modern art tool. *BRAIN. Broad Research in Artificial Intelligence and Neuroscience*, 14(2):115–127.

Cohen-Or, D., Chrysanthou, Y., Durand, F., Greene, N., Koltun, V., and Silva, C. (2000). Visibility, problems, techniques and applications. *ACM SIGGRAPH Course*, 4.

Colubri, A. (2023). Getting started with the android mode. In *Processing for Android: Create Mobile, Sensor-aware, and XR Applications Using Processing*, pages 3–16. Springer.

Colubri, A. and Fry, B. (2012). Introducing processing 2.0. In *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2012, Los Angeles, California, USA, August 5-9, 2012, Talks Proceedings*, page 12. ACM.

Donaldson, A. F., Evrard, H., Lascu, A., and Thomson, P. (2017). Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29.

Ferraz, O., Menezes, P., Silva, V., and Falcão, G. (2021). Benchmarking vulkan vs opengl rendering on low-power edge gpus. In *International Conference on Graphics and Interaction, ICGI 2021, Porto, Portugal, November 4-5, 2021*, pages 1–8. IEEE.

Fry, B. (2008). *Visualizing data - exploring and explaining data with the processing environment*. O'Reilly.

Fry, B. (2021). Changes in v4.0.

Gfrerer, T. (2025). Openframeworks website - introducing opengl for of. Available at https://poniesandlight.co.uk/reflect/of-vulkan-renderer-update/.

Kenwright, B. (2017). Getting started with computer graphics and the vulkan API. In Bednarz, T., editor, *SIGGRAPH Asia 2017 Courses, Bangkok, Thailand, November 27 - 30, 2017*, pages 5:1–5:86. ACM.

Kilgard, M. J. (1997). Realizing opengl: two implementations of one architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 45–55.

Kilgard, M. J. and Akeley, K. (2008). Modern opengl: its design and evolution. In *ACM SIGGRAPH ASIA 2008 courses*, pages 1–31.

Kligge, M. (2024). Comparison of webgl and webgpu as alternatives for implementing gpgpu computing in the browser. *Abschlussbericht FEP 2023/2024*, page 13.

Lapinski, P. (2017). *Vulkan cookbook*. Packt Publishing Ltd.

Linares-Pellicer, J., Santonja-Blanes, J., Tormos, P. M., and Cuesta-Frau, D. (2009). Using processing.org in an introductory computer graphics course. In Domik, G. and Scateni, R., editors, *30th Annual Conference of the European Association for Computer Graphics, Eurographics 2009 - Education Papers, Munich, Germany, March 30 - April 3, 2009*, pages 23–28. Eurographics Association.

Lujan, M., Baum, M., Chen, D., and Zong, Z. (2019). Evaluating the performance and energy efficiency of opengl and vulkan on a graphics rendering server. In *International Conference on Computing, Networking and Communications, ICNC 2019, Honolulu, HI, USA, February 18-21, 2019*, pages 777–781. IEEE.

Lujan, M., McCrary, M., Ford, B. W., and Zong, Z. (2021). Vulkan vs opengl ES: performance and energy efficiency comparison on the big.little architecture. In *IEEE International Conference on Networking, Architecture and Storage, NAS 2021, Riverside, CA, USA, October 24-26, 2021*, pages 1–8. IEEE.

Nilssen, M. G. B. (2023). Vulkan port of opengl-based cuda snow simulation. Master's thesis, NTNU.

Nott, G. (2013). Opengl multi-context fact sheet. perfect internal disorder. [blog]. Available at https://blog.gvnott.com/some-usefull-facts-about-multipul-opengl-contexts/.

Pankratz, D., Nowicki, T., Eltantawy, A., and Amaral, J. N. (2021). Vulkan vision: Ray tracing workload characterization using automatic graphics instrumentation. In *2021*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE.

Reas, C. and Fry, B. (2007). Processing: a programming handbook for visual designers and artists. In *Processing: a programming handbook for visual designers and artists*, pages 1–8. Mit Press.

Sandberg, E. (2019). Creative coding on the web in p5. js: a library where javascript meets processing.

Sellers, G., Wright Jr, R. S., and Haemel, N. (2013). *OpenGL superBible: comprehensive tutorial and reference*. Addison-Wesley.

Shiraef, J. A. (2016). An exploratory study of high performance graphics application programming interfaces.

Shreiner, D. et al. (2009). *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education.

Singh, P. (2016). *Learning Vulkan*. Packt Publishing Ltd.

Szabó, D. and Illés, Z. (2020). Vulkan in c# for multi-platform real-time graphics. In *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 687–692. IEEE.

Szabó, D. and Illés, Z. (2022). Real-time rendering with opengl and vulkan in c#. In *Recent Innovations in Computing: Proceedings of ICRIC 2021, Volume 2*, pages 599–611. Springer.

Tolo, L. O., Viola, I., Geitung, A. B., Soleim, H., and Patel, D. (2018). Multi-gpu rendering with the open vulkan API. In *31st Norsk Informatikkonferanse, NIK 2018, Universitetet i Oslo, Oslo, Norway, September 18-20, 2018*. Bibsys Open Journal Systems, Norway.

Unterguggenberger, J., Kerbl, B., and Wimmer, M. (2022). The road to vulkan: Teaching modern low-level apis in introductory graphics courses. *The Eurographics Association*.

Unterguggenberger, J., Kerbl, B., and Wimmer, M. (2023). Vulkan all the way: Transitioning to a modern low-level graphics API in academia. *Comput. Graph.*, 111:155–165.

Varanka, S. (2023). Platform-agnostic data visualization in qt framework. Master's thesis, S. Varanka.

Wolff, D. (2011). *OpenGL 4.0 shading language cookbook*. Packt Publishing Ltd.