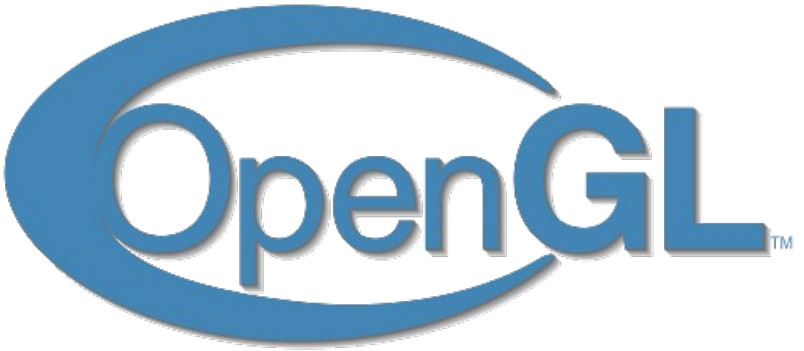
Software practical final presentation

Niels Buwen
David Sprengel

Vulkan vs OpenGL

Conceptual differences and performance



'Vulkan is not well-suited to simple test applications;
neither is it a suitable aid for teaching graphics concepts.'

– Graham Sellers, Vulkan Programming Guide

Outline

- Introduction: goal and history
- Similarities
- Conceptual differences
- Getting started
- Performance
- Summary
- Demonstration (video)

Introduction

What are OpenGL and Vulkan?

- API for 3D graphics applications
- Platform-independent
- Programming language-independent
- Developed by the Khronos Group[1]

1 https://www.khronos.org/

Goal

1) Develop minimal OpenGL and Vulkan program
2) Implement Exercises from 'Computergrafik I'[1] and 'Computergraphics'[2] in OpenGL and Vulkan
3) Compare results
   - Performance
   - Programming experience
   - Visuals

1 Lecture by Susanne Krömker
2 Lecture by Filip Sadlo - https://vcg.iwr.uni-heidelberg.de/teaching/2016-17/cg/

History - OpenGL

- Introduced in 1992
- Used fixed-function-pipeline (`glBegin()`,`glEnd()`)
- In 2008: Version 3.0 with programmable-pipeline (shaders)
- 2012: Version 4.3 with compute shaders
- Supports: Linux, macOS, Windows(but DirectX is more common)
- Separate API for Android and iOS (OpenGL ES)

History - Vulkan

- Created in 2014 as 'glNext'
- From the very start only programmable-pipeline and computing capabilities
- Announced in 2015 at GDC
- 26.02.2018: macOS and iOS support through MoltenVK
- Supports: Windows, Linux, macOS, Android, iOS

Similarities

- Graphics/compute APIs
- Programmable pipeline
- Configurable pipeline
- Cross-platform
- GLSL/SPIR-V

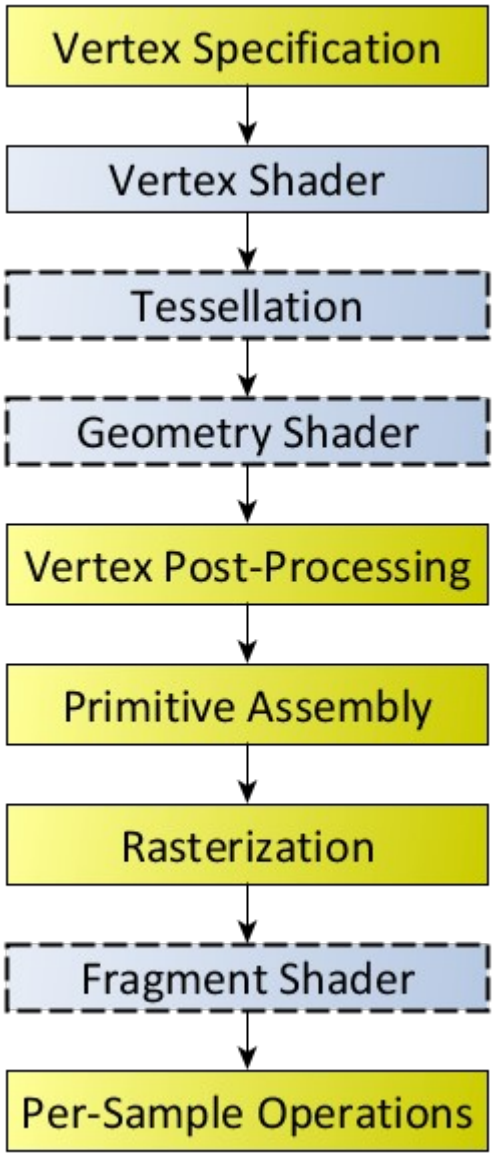


Figure: Graphics pipeline

Conceptual Differences

OpenGL	Vulkan
Global state machine (`glClearColor()`)	Object oriented local state (`VkInstance`)
Everything is preconfigured	Must configure **everything!** (depth resources, pipeline, …)
Dynamic (can change shaders, pipeline parameters at runtime)	Static (must recreate pipeline to change anything)
Automatic memory management (`glGenBuffers()`)	Manual memory management (buffers, images)
Render loop: '*foreach* object *do* draw()'	Setup: 'record draw calls' Render loop: 'replay draw calls'
Focus on graphics	Unified management of compute kernels and graphical shaders

Minimal Project – OpenGL

- Update GPU driver!
- Install dev libs (xorg-dev)
- Setup:
  - Window (3rd party library, e.g. GLFW)
  - Compile/link shaders
  - Minimal configuration (clear color)
  - Prepare data (VAO, VBO)

Minimal Project – OpenGL /2

- Render loop
  - Clear buffers (color buffer, z-buffer)
  - For each object:
    - Use shader
    - Update uniforms
    - Issue draw call
- < 250 LoC

Minimal Project – Vulkan

- Update GPU driver!
- Install dev libs
- Install SDK[1]
- Setup:
  - Window, create Instance, choose device, create framebuffers, create renderpass, create command buffers, create sync primitives, create swapchain, …, many things ~1800 LoC

1 https://vulkan.lunarg.com/

Minimal Project – Vulkan /2

- Record command buffers
  - For each object: record draw call
  - Record clear operations
- Render loop
  - Aquire swapchain image (Extension)
  - Replay command buffers
  - Submit swapchain image (Extension)

Render loop – OpenGL

```
// Render loop
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glBindVertexArray(VAO);
glUseProgram(SHADER);
glDrawElements(MODE, COUNT, ...);
glfwSwapbuffers(WINDOW);
```

Render loop – Vulkan

```
vkQueueWaitIdle(QUEUE);
vkAcquireNextImageKHR(DEVICE, SWAPCHAIN, &IMAGE);
VkSubmitInfo submit;
VkPipelineStageFlags wait[] =
   {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT}
submit.waitSemaphoreCount = 1;
submit.pWaitSemaphores = &IMAGE_AVAILABLE;
submit.pWaitDstStageMask = wait;
```

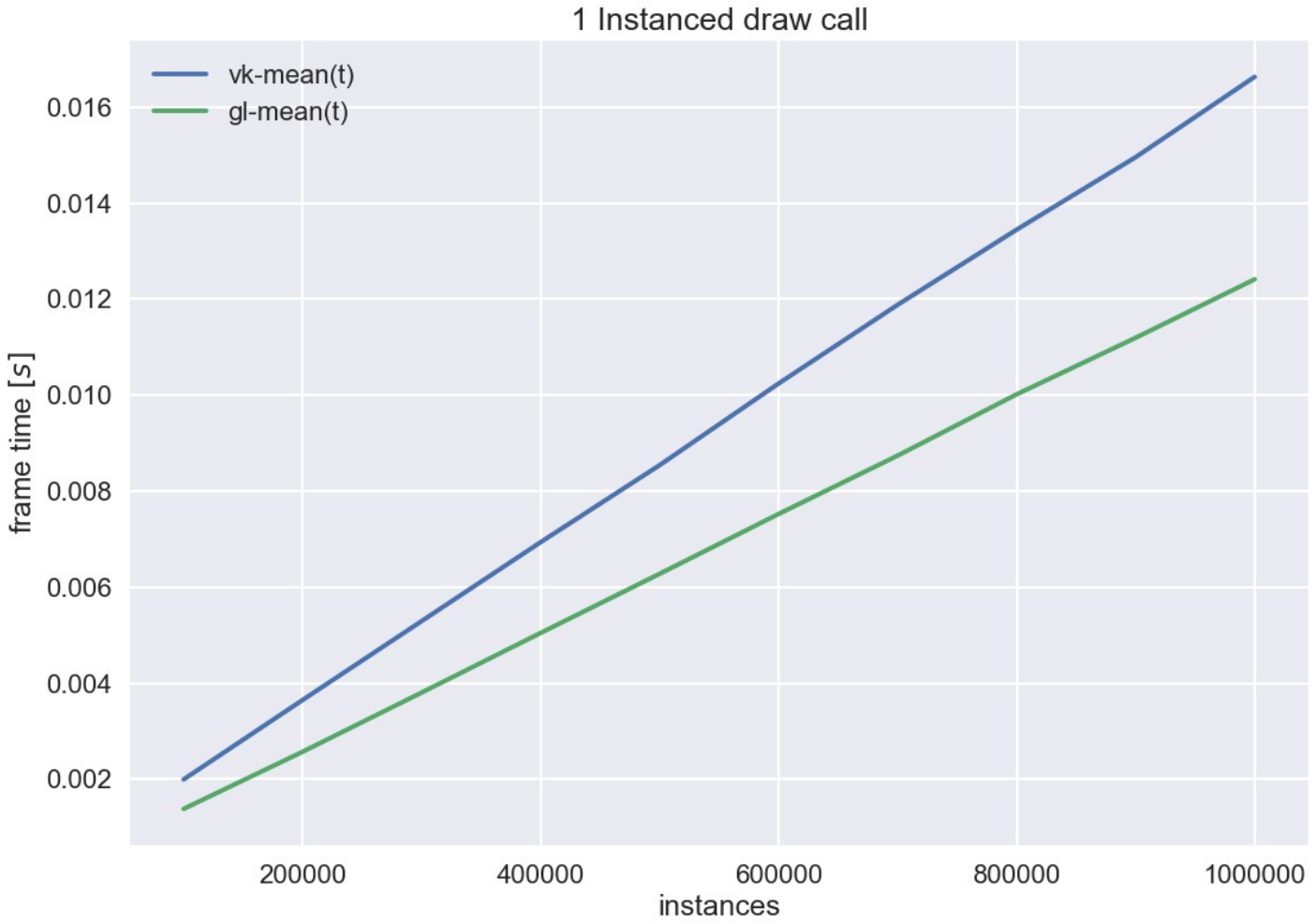
Render loop – Vulkan /2

```
submit.commandBufferCount = 1;
submit.pCommandBuffers = &COMMAND_BUFFER;
submit.signalSemaphoreCount = 1;
submit.pSignalSemaphores = &RENDER_FINISHED;
VkPresentInfoKHR present;
present.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
present.waitSemaphoreCount = 1;
present.pWaitSemaphores = &RENDER_FINISHED;
present.swapchainCount = 1;
```

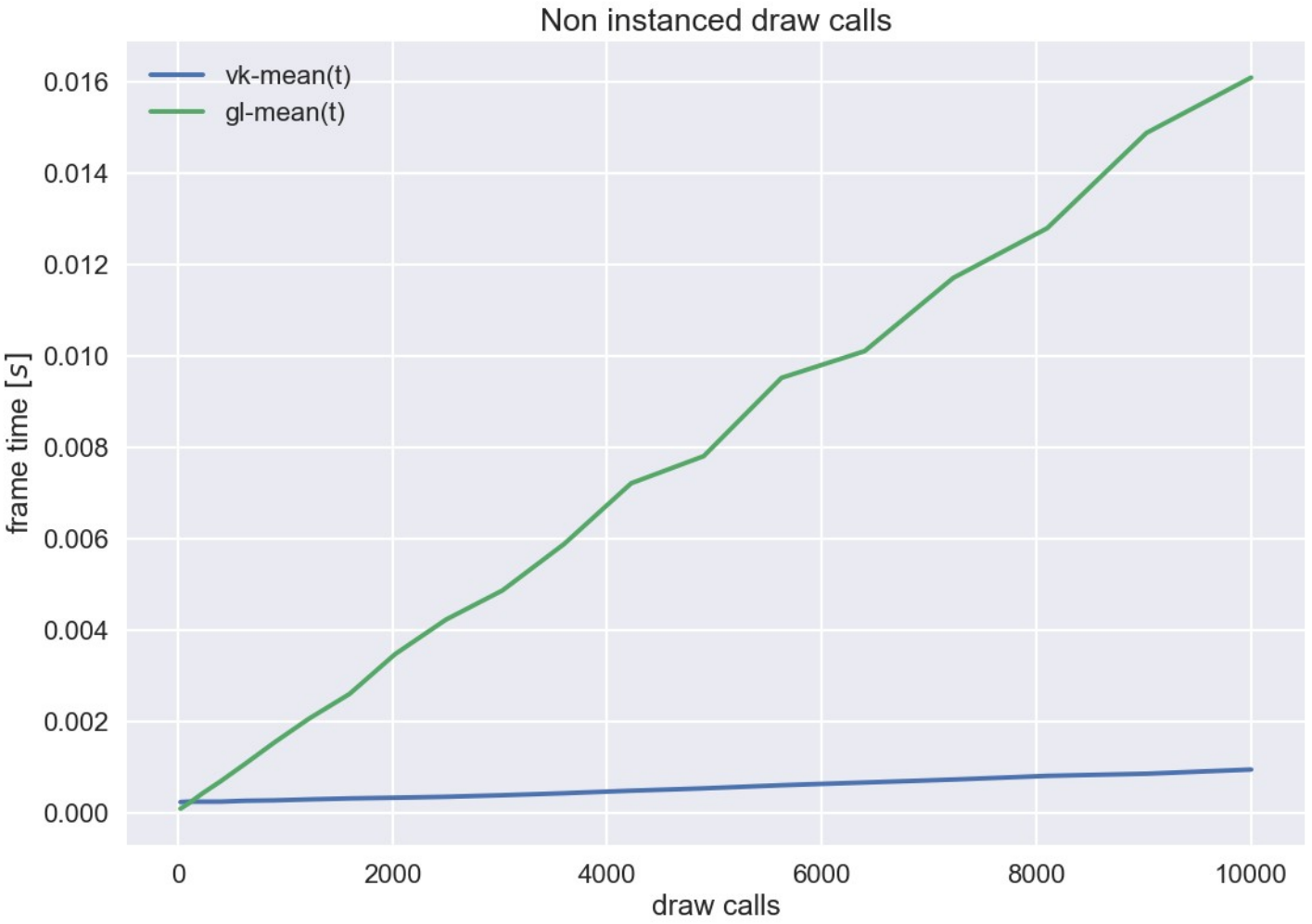
Render loop – Vulkan /3

```
present.pSwapchains = &SWAPCHAIN;
present.pImageIndices = &IMAGE;
present.pResults = nullptr;
// actual command to draw something
vkQueuePresentKHR(QUEUE, &present)
```

Performance – instanced



Performance – not instanced



Performance – Vulkan

There are some cases where Vulkan is faster

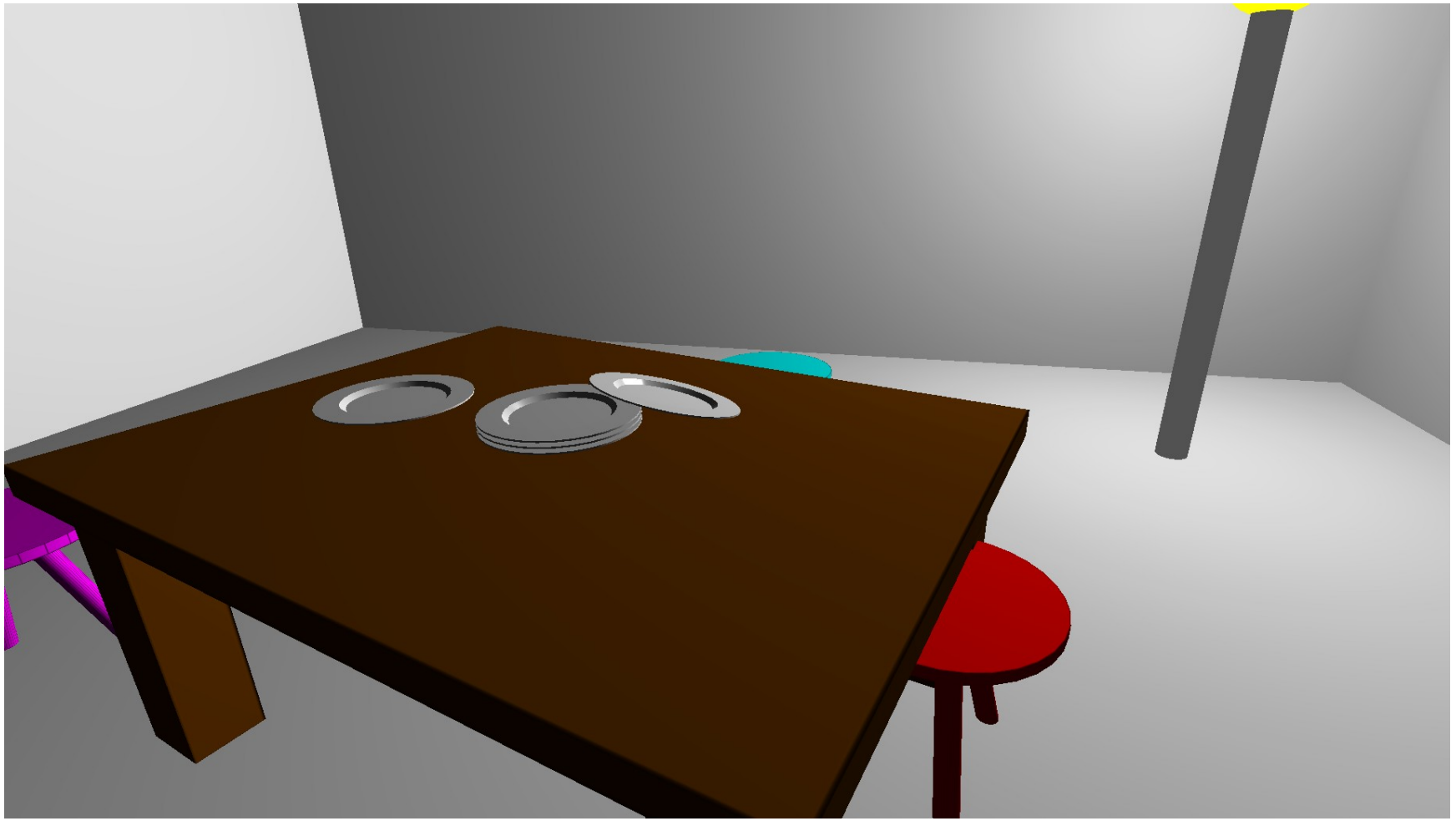
- Many draw calls vs many instances
- Hardware dependent (nVidia vs AMD)
- Parallel command buffer creation
  - Cannot concurrently issue OpenGL draw calls

Summary

- Start with OpenGL!
- Vulkan is more verbose
  - Could be solved by nVidia's vkHLF[1]
- No visual differences (so far)
- Vulkan is faster in some specially designed cases
- Lua[2] is our favourite embedded scripting language

1 Vulkan High Level Framework https://github.com/nvpro-pipeline/VkHLF
2 Lua programming language https://www.lua.org/

Demo



Questions

?

?

?