Future of VR with Vulkan

JANG DAEMYUNG
Engineer
Samsung Electronics

Who we are?

Agenda

- Introduction
- Why Vulkan?
- Technical Details
- Case Study

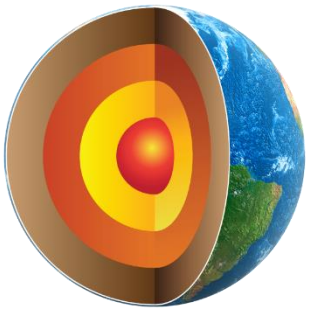
Introduction

Introduction

History

- The Force Awakens: October 2012
  - glCommon TSG formed to consider redesign of OpenGL / ES
  - Brainstorming and design sketches

- A New Hope: June / July 2014
  - Effort rebooted as glNext – becomes the top priority
  - Unprecedented participation from key ISVs
  - AMD donates Mantle as a starting point



- Renamed and disclosed at GDC 2015

- Public Launch on February 16th, 2016

Vulkan™

Introduction

Vulkan vision and goals at project launch

- An open-standard, cross-platform graphics+compute API
  - Compatibility break with OpenGL
  - Start from first principles

- Goals
  - Clean, modern architecture
  - Multi-thread / multicore-friendly
  - Greatly reduced CPU overhead
  - Full support for both desktop and mobile GPU architectures
  - More predictable performance – no driver magic
  - Improved reliability and consistency between implementations

Introduction

Wide industry support

* Image from Khronos 3D BoF of GDC 2016

- A whole industry, working together
  - GPU and SoC Vendors
  - Game and middleware developers
  - Platform owners, Content providers

- All Khronos resources are open source
  - http://github.com/KhronosGroup/

Introduction

Brought to you by…

The Vulkan and SPIR-V working groups, Seattle, January 2016

* Image from Khronos Group Flickr

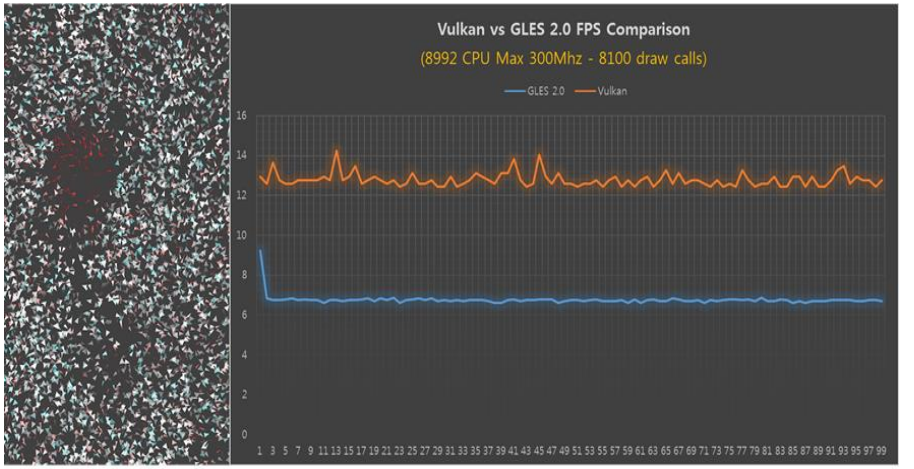
A bunch of people who really care about open graphics future

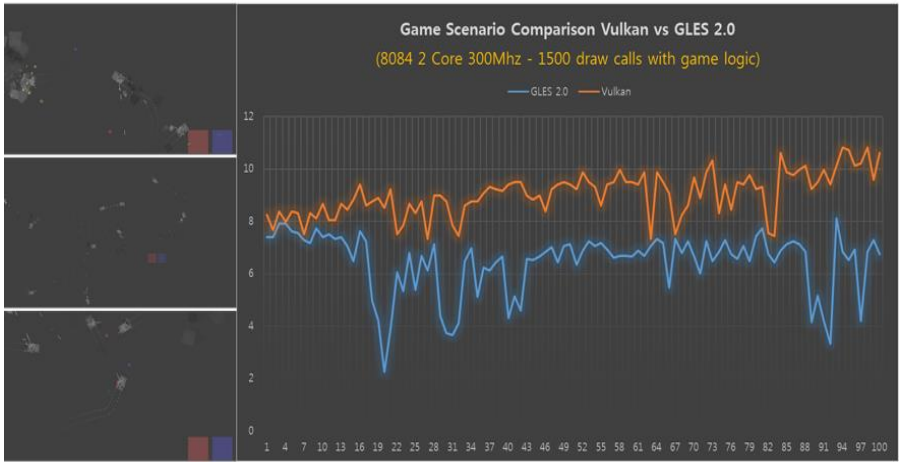
Why Vulkan?

Why Vulkan?

Super efficient

- Did many investigation from the very early stage like mid 2015
- Limitless draw calls and render passes now allowed in mobile product
- Simple 3D scenes show more than 5X CPU offload from GPU
- It gives real gains like 2X FPS with some scenes

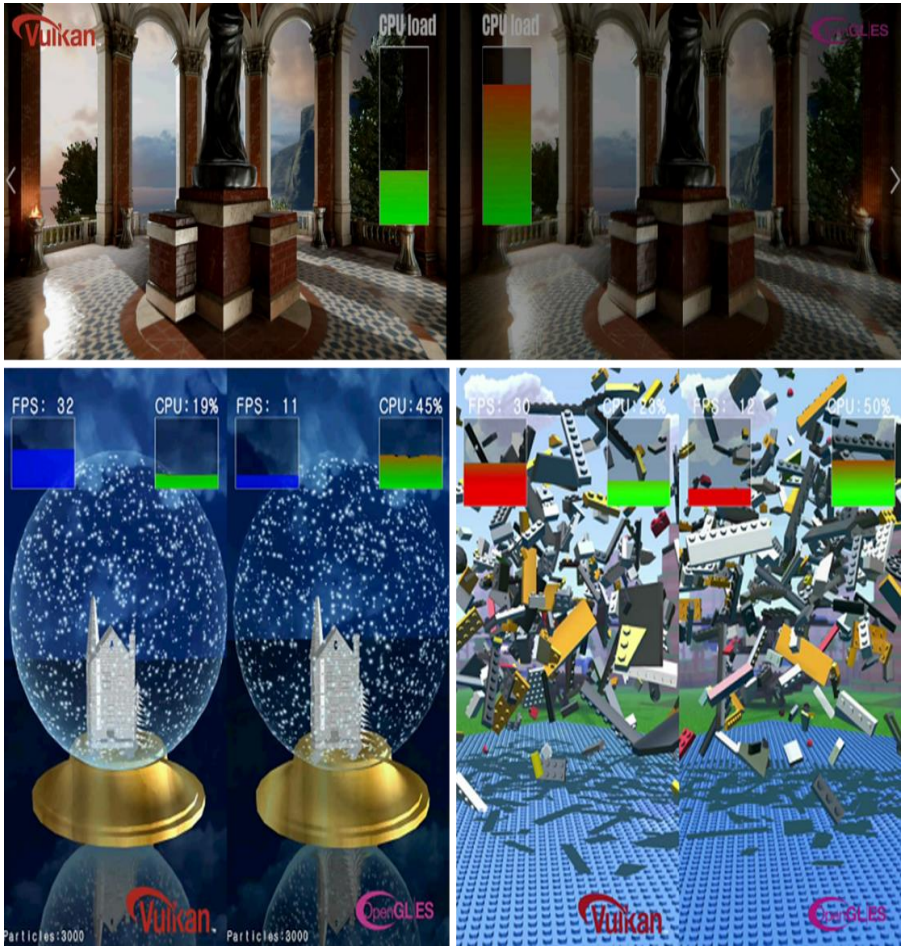




Why Vulkan?

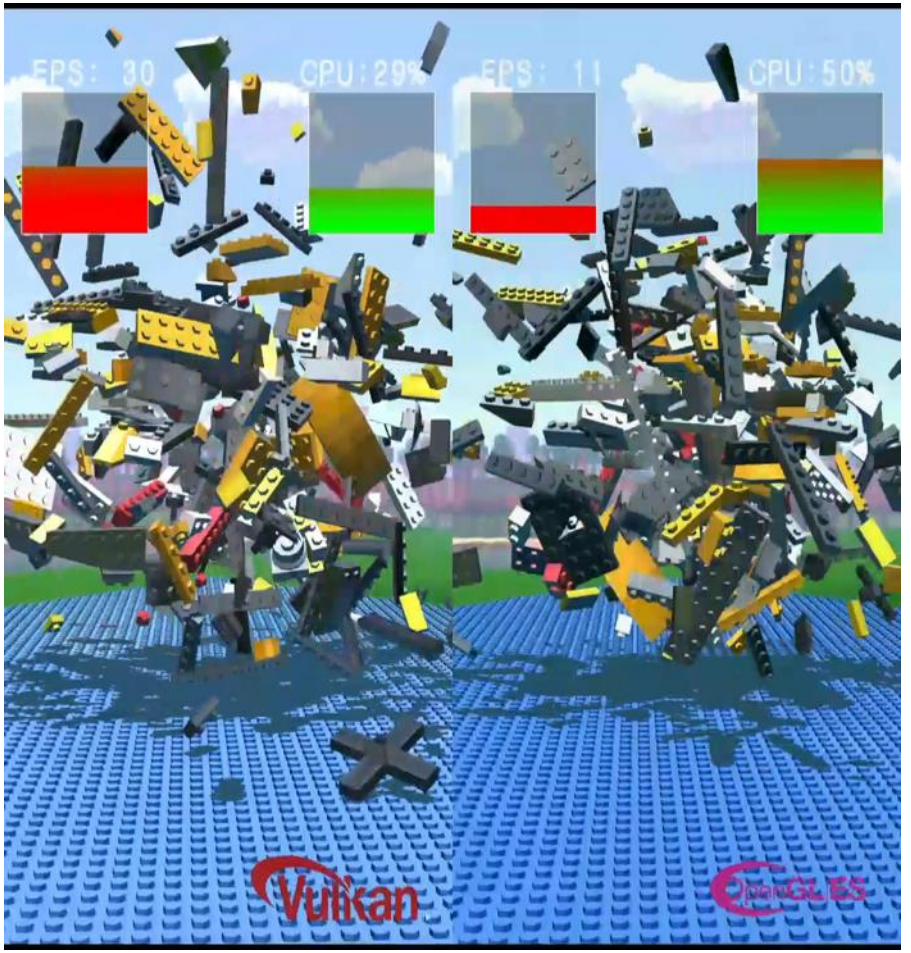
Real beneficial

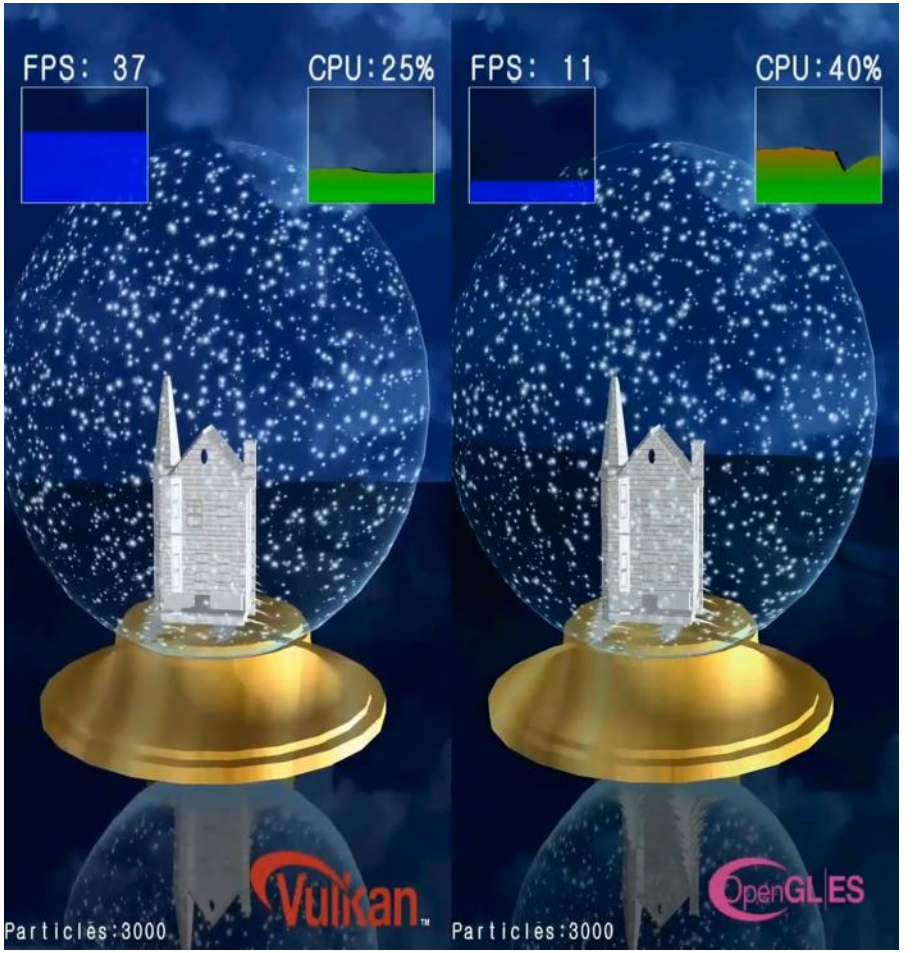
- More performance or less power / thermal
- More visual effect and post processing can be covered within same hardware resources
- Various explicit ways to optimize application
- Means you can make your game runs faster and look better



Why Vulkan?

Real Beneficial





Why Vulkan?

Open, Open and Open!

- Open standard
- Cross platform
- Open source
- Why Samsung chose Vulkan?

  Because an open dev model is the best to support developers.
  This will help to accelerate adoption of Vulkan to the ecosystem and surely can make a healthy mobile environment.

Why Vulkan?

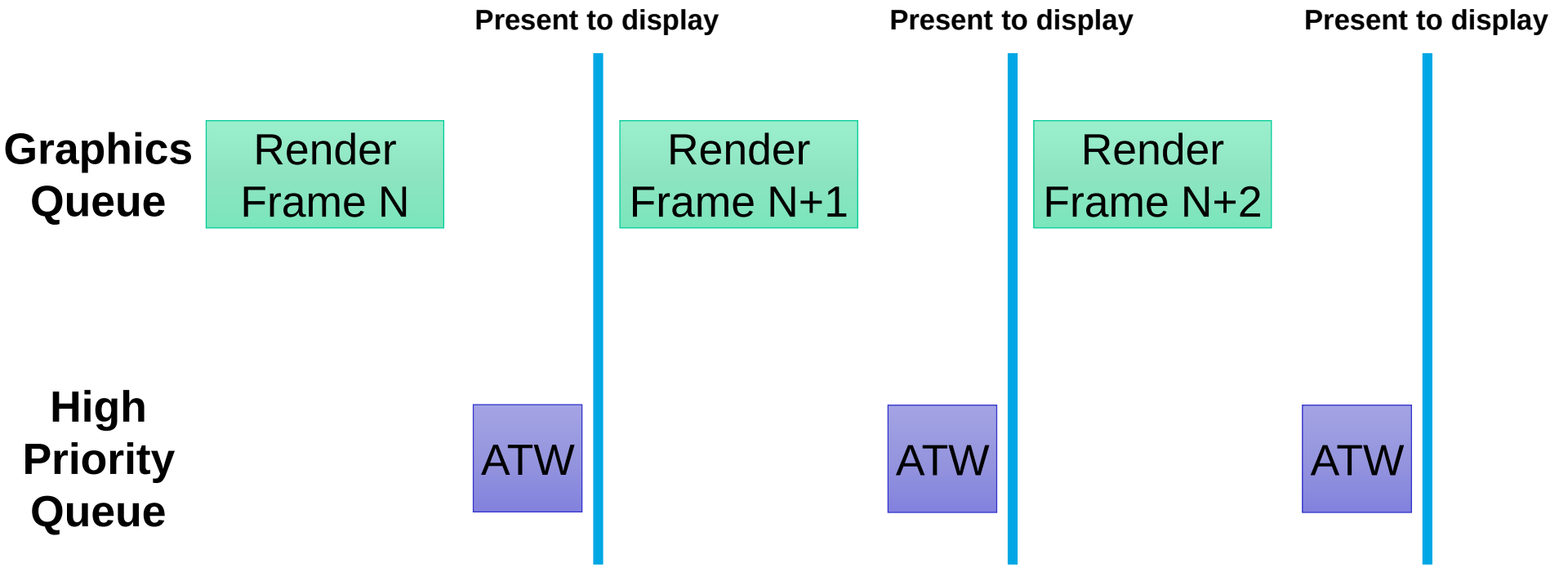
Latency at VR

- The time interval between initiating an action and rendering
- High latency can lead to uncomfortable psychophysical effects
- Stall GPU because busy CPU
- Mostly 60 FPS is not enough and solid 60 FPS is a minimum requirement

Why Vulkan?

Asynchronous Time warp

- High priority queue for the time-critical task

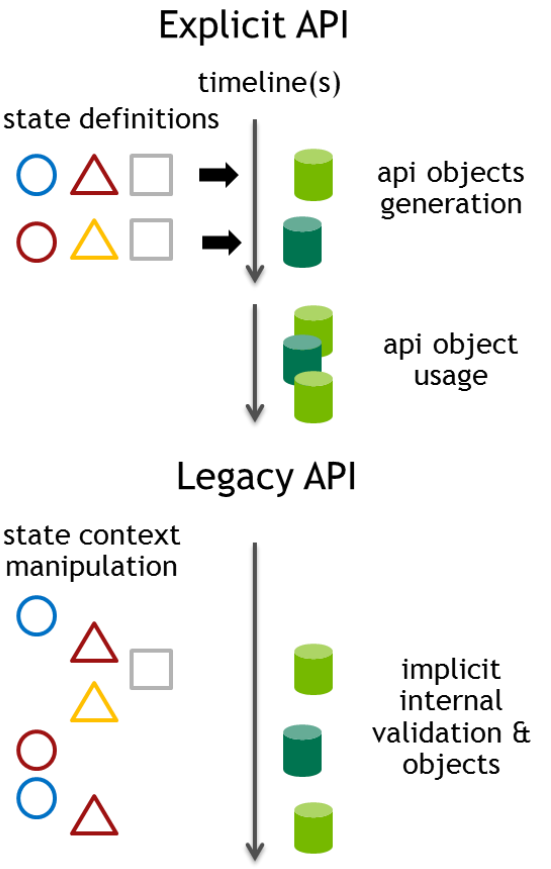


Technical Details

Explicit API

What it is?

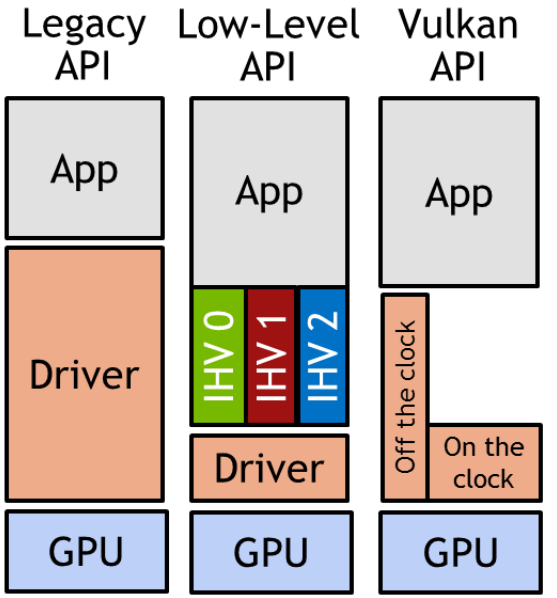
- Providing information at the right time
- Predictable performance costs
  - Creating pipelines, allocating memory, more…
- No driver magic on the clock
  - Remove guesswork and late decision making
- Simpler drivers
- Better scheduling over CPU & GPU work



Explicit API

What it is not?

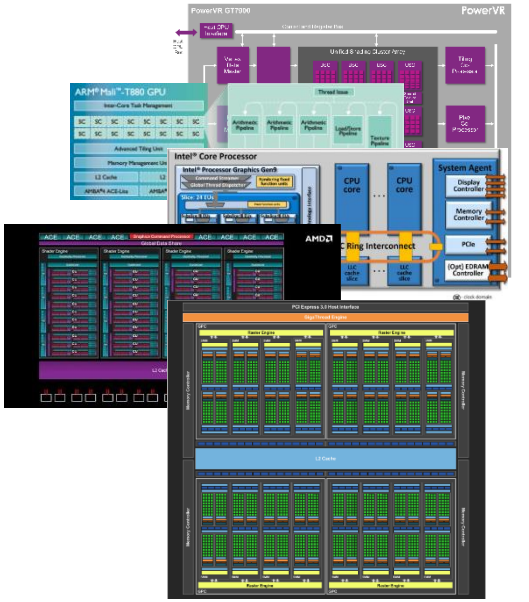
- Low-level == Thin layer over specific HW implementation, little abstraction
  - Not possible given wide variety of hardware
- Making everything the app's problem
- Getting the driver out of the way
- Solves a different problem than we were asked to



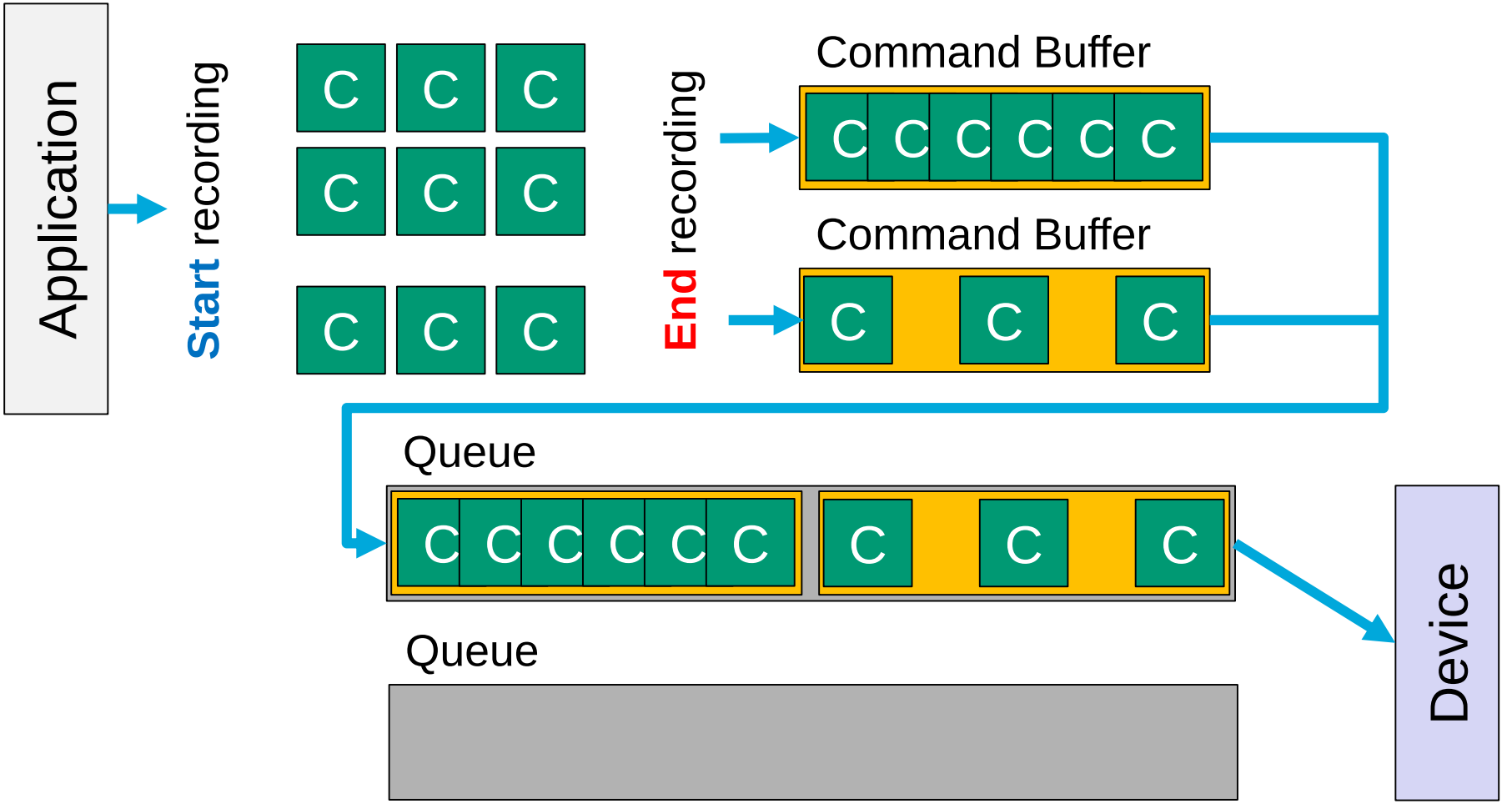
Portability

Write once, run anywhere

- Strong desire to avoid forking the ecosystem
- A single API(desktop, mobile)
- Supports various GPU hardware(IMR, TBR, TBDR)



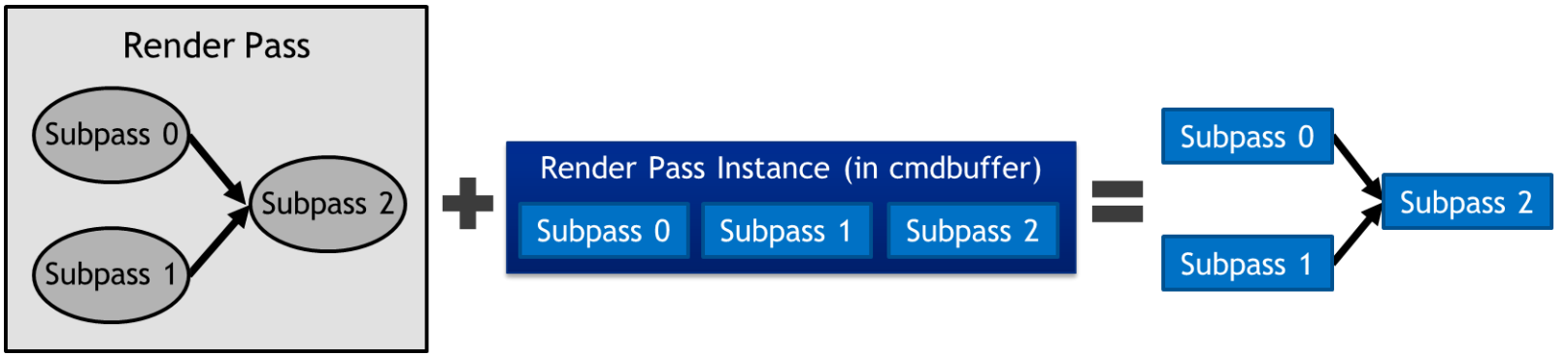
Execution Model



Multipass

- Evolved into supporting multiple “subpasses”

- A dependency graph (DAG) between subpasses
  - Each Node is a subpass – a list of attachments with format info
  - Each Edges is an execution/memory dependency between subpasses
  - Each edge indicates whether the dependency is tiler-friendly



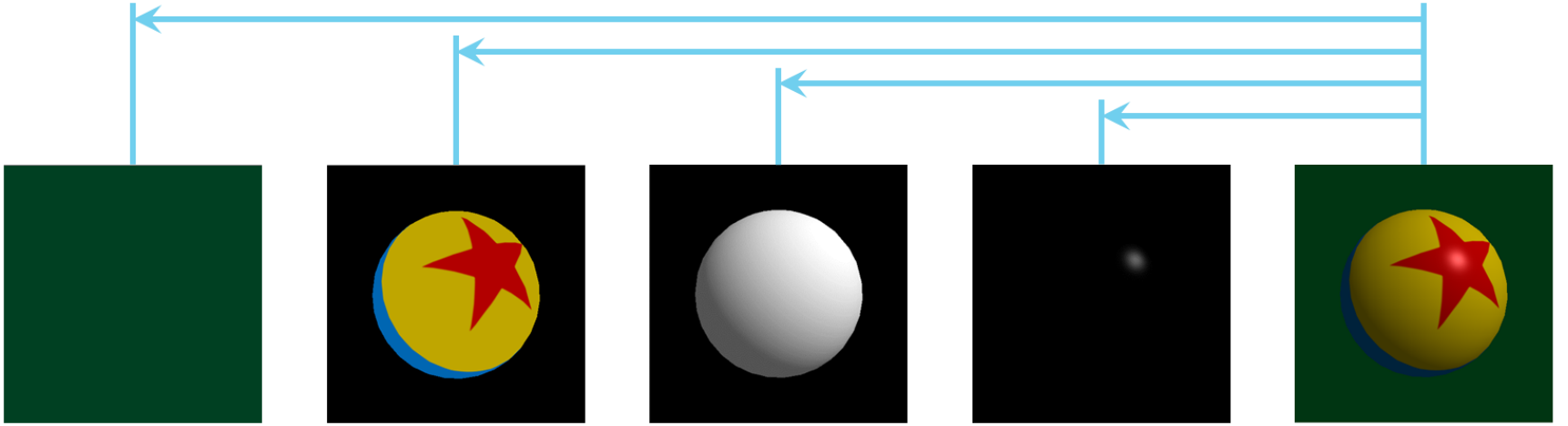
Multipass

Tiling

RenderPass – Subpass Dependencies

Background

Final = ((Surface * AmbientDiffuse) + Specular) Over



Subpass0
Background

Subpass1
Surface

Subpass2
AmbientDiffuse

Subpass3
Specular

Subpass4
Final

Comparison

OpenGL|ES and Vulkan

Issue	Naïve GL	Vulkan
Deterministic state validation/pre-compilation	no	Yes
Improved single thread performance	no	Yes
Multi-threaded work creation	no	yes
Multi-threaded work submission (to driver)	no	yes
GPU based work creation	no	partial (through MDI)
Ability to re-use created work	no	yes
Multi-threaded resource updates	no	Yes
Learning curve	low	Significant
Effort	low	Significant

Unlikely to Benefit

- Scenarios to reconsider coding to Vulkan
  - Need to compatibility to pre-Vulkan platform
  - Heavily GPU-bound application
  - Heavily CPU-bound application due to non-graphics work
  - Single-threaded application, unlikely to change
  - App can target middle-ware engine, avoiding 3D graphics API dependencies(Consider using an engine targeting Vulkan, instead of coding Vulkan yourself)

Conclusion

- Early concerns about being too hard to use / only viable for AAA developers
  - But in the end, the abstractions we have are quite usable and the API evolved being “simpler” than first versions

- Higher-level abstractions allow for better hardware innovations, optimization opportunities, and implementation on a wide variety of hardware

- Not “Low-Level” – careful design decisions give apps the tools they need to improve performances

* Contents from Khronos 3D BoF of GDC 2016

Case Study

Goals & Targets

- **Goals**
  - Vulkan backend for existing graphics engine
  - **Same apps sources** → multiple pipelines (VK/GL)
  - **Minimal effort for apps migration** to Vulkan

- **Target Apps**
  - VR 1st person shooter show-case

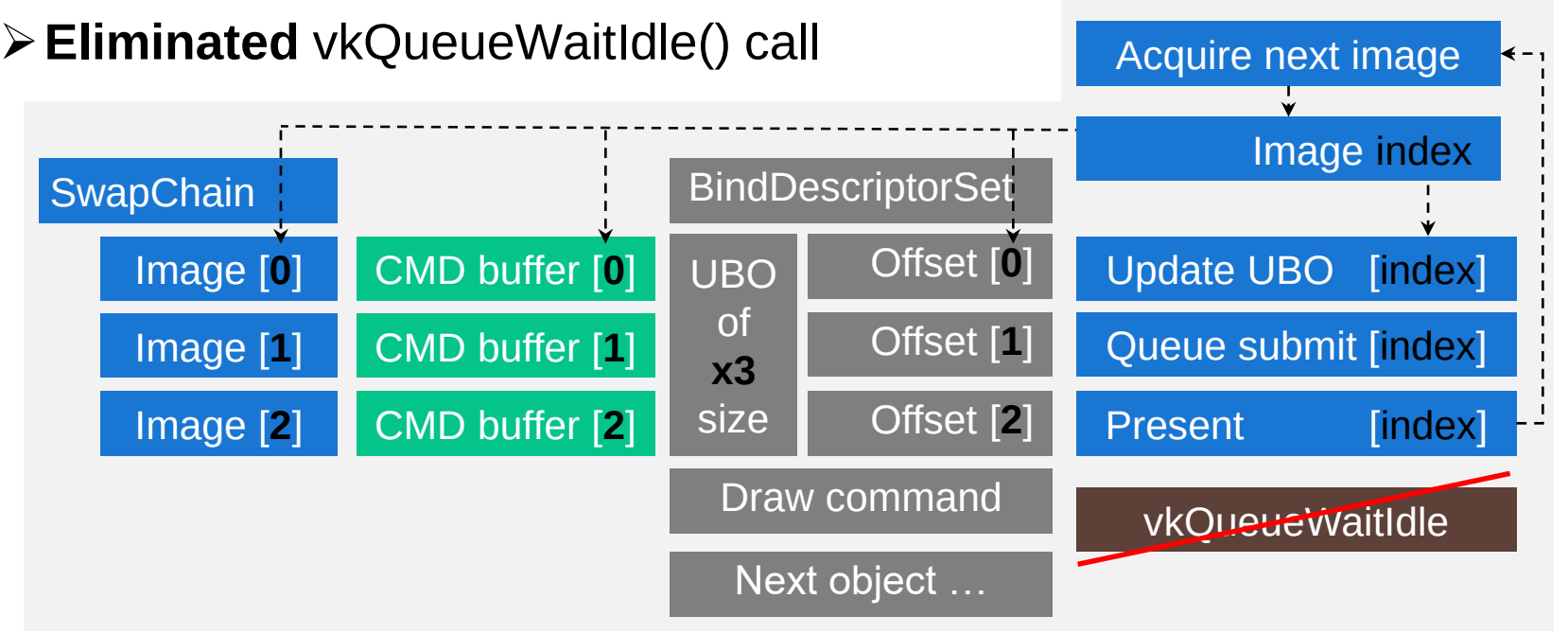
Game Case

VR shooter

Game Case

Static scene(700 draw calls)

- Triple buffering for command buffers and UBOs (once recorded buffers)
- **Eliminated** vkQueueWaitIdle() call

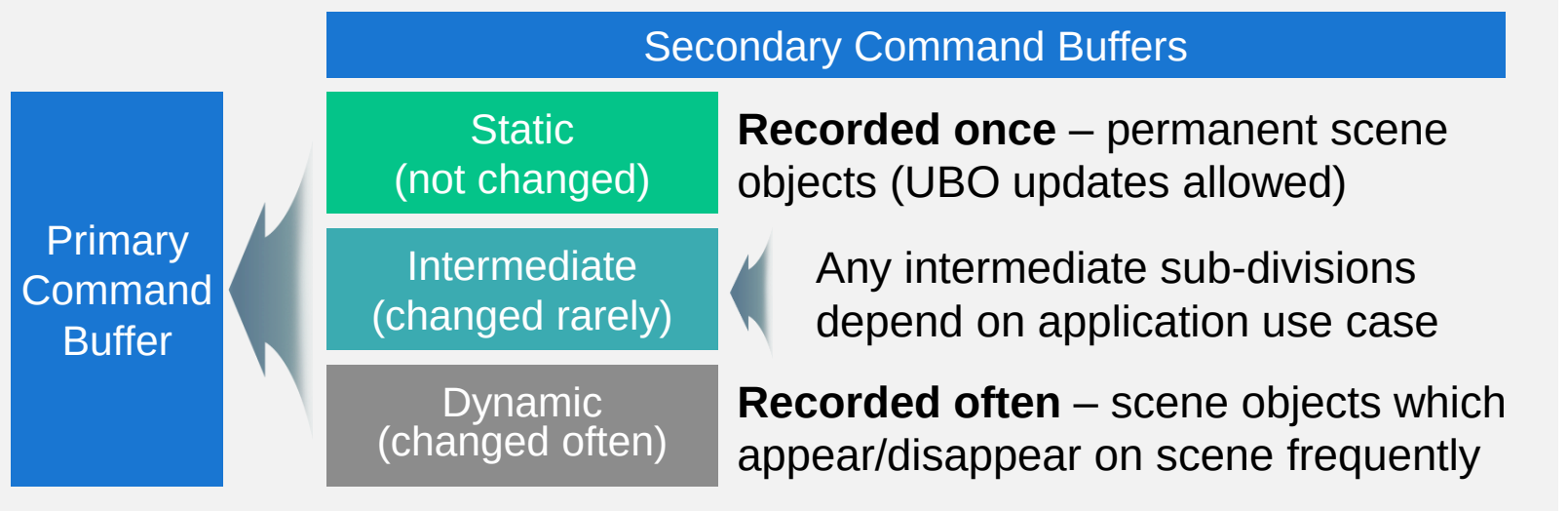


- Vulkan performance ~40% better than GL on low CPU clock(< 900 MHz)
- Performance gap decreases on higher CPU clock
- “Static” scene NOT applicable for real use-case

Game Case

Dynamic scene(700 draw calls)

- Recording command buffers for each frame
- Scene divided on “Static” and “Dynamic” secondary command buffers
- Single-thread secondary buffers recording and UBO updates

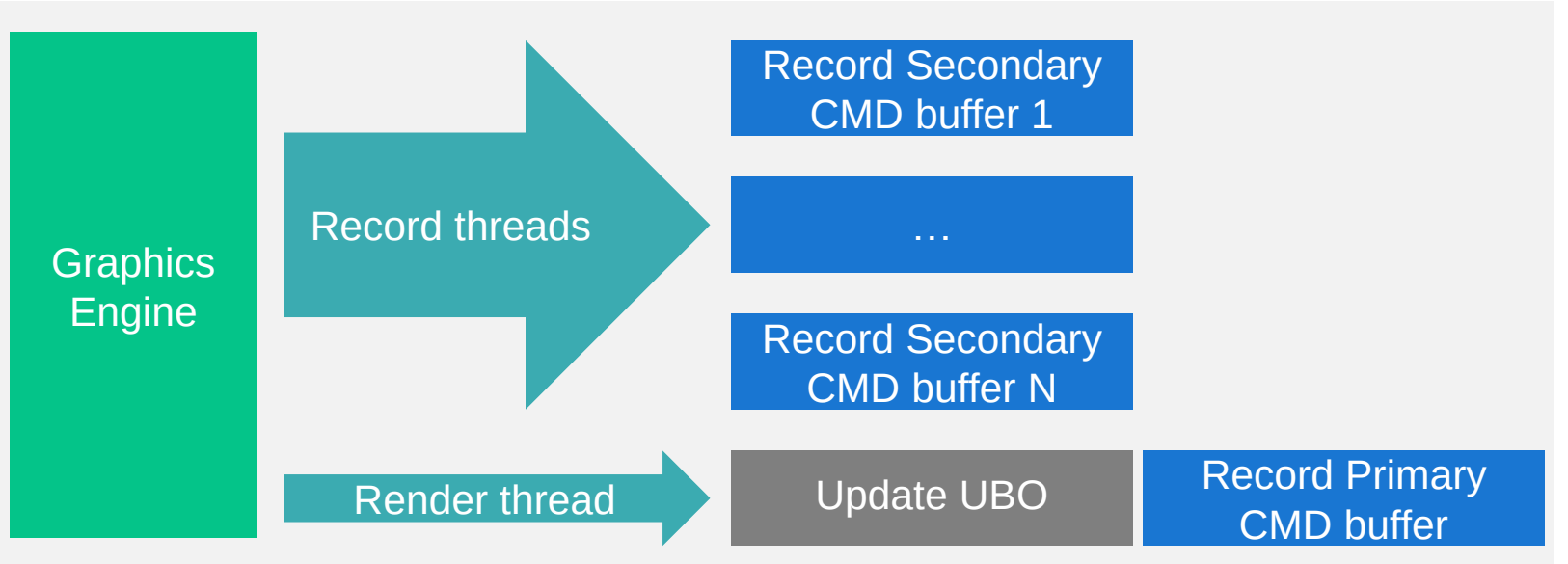


- Performance ~25% less than for “Static” scene (CPU-bounded)
- Applicable for real use case, but optimization required…

Game Case

Multi threaded dynamic scene(700 draw calls)

- Multi-thread secondary command buffers recording
- UBO updates in parallel with buffers recording



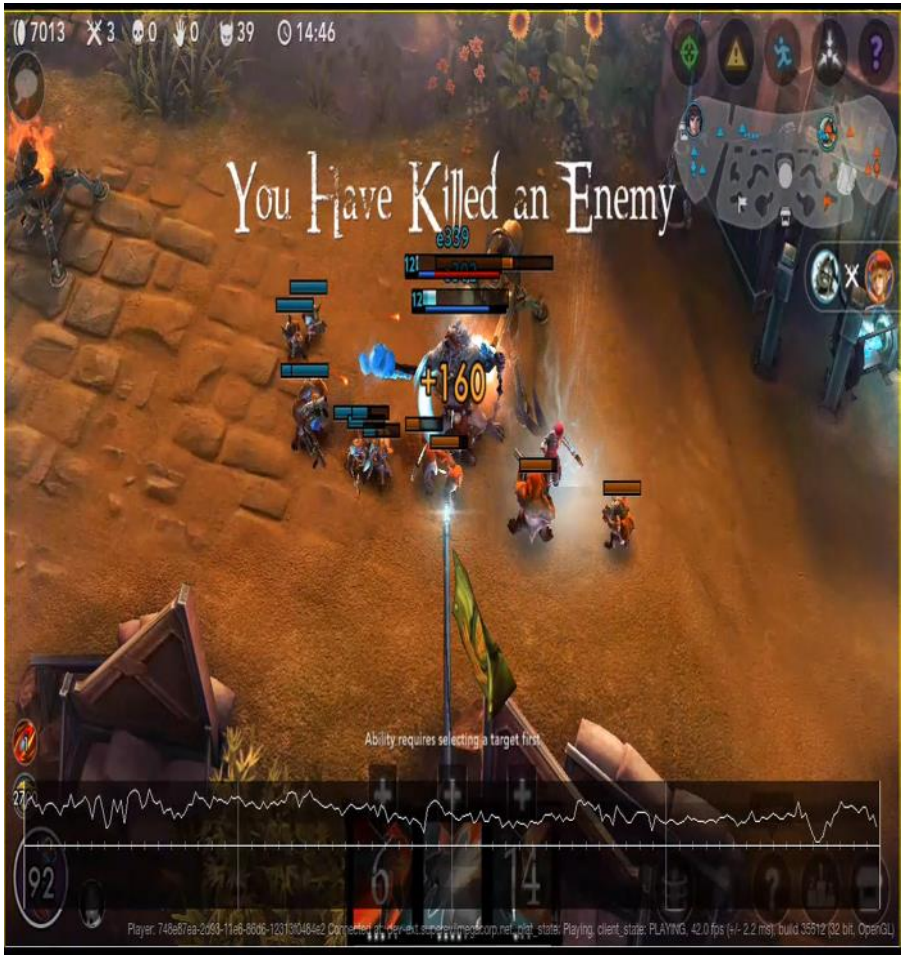
- Performance ~10% less than for “Static” scene
- Applicable for real use case

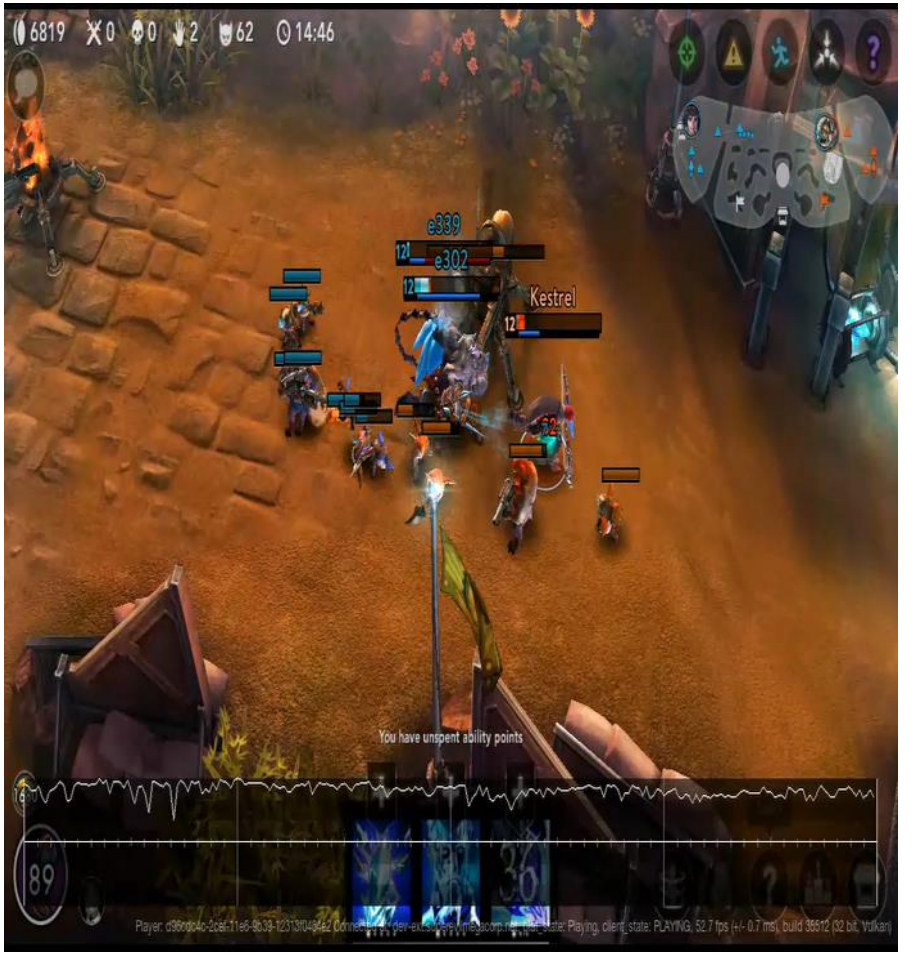
Game Case

Unreal Engine

Game Case

VainGlory





Game Case

Need For Speed


SAMSUNG