KHRONOS GROUP

Vulkan Update

CEDEC 2016
Computer Entertainment Developers Conference

Neil Trevett, NVIDIA | Khronos President
ntrevett@nvidia.com | @neilt3d

Session Speakers

Neil Trevett	NVIDIA	Vulkan Ecosystem
Jung Woo Kim	Samsung	Advanced Mobile Gaming with Vulkan
Pierre Rahier	Silicon Studio	Porting Xenko Engine to Vulkan
Eisaku Ohbuchi	DMP	DMP and Vulkan

KHRONOS™ GROUP

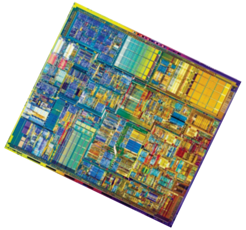
Khronos Mission

PROMOTER MEMBERS

Over 130 members worldwide
Any company is welcome to join

Software

Silicon



Khronos is an International Industry Consortium of over 100 companies creating royalty-free, open standard APIs to enable software to access hardware acceleration for 3D graphics, parallel computing and vision processing

The Need for a New Generation GPU API

- **Explicit**
  - Direct GPU control, predictable - no driver surprises
- **Faster**
  - Reduce CPU overhead and latency, multi-thread scaling
- **Portable**
  - Efficient on cloud, desktop, console, mobile and embedded

OpenGL has evolved over 25 years and continues to meet industry needs - but there is a need for a complementary API approach

GPUs are increasingly programmable and compute capable + platforms are becoming mobile, memory-unified and multi-core

GPUs will accelerate graphics, compute, vision and deep learning across diverse platforms: FLEXIBILITY and PORTABILITY are key

Three New Generation GPU APIs

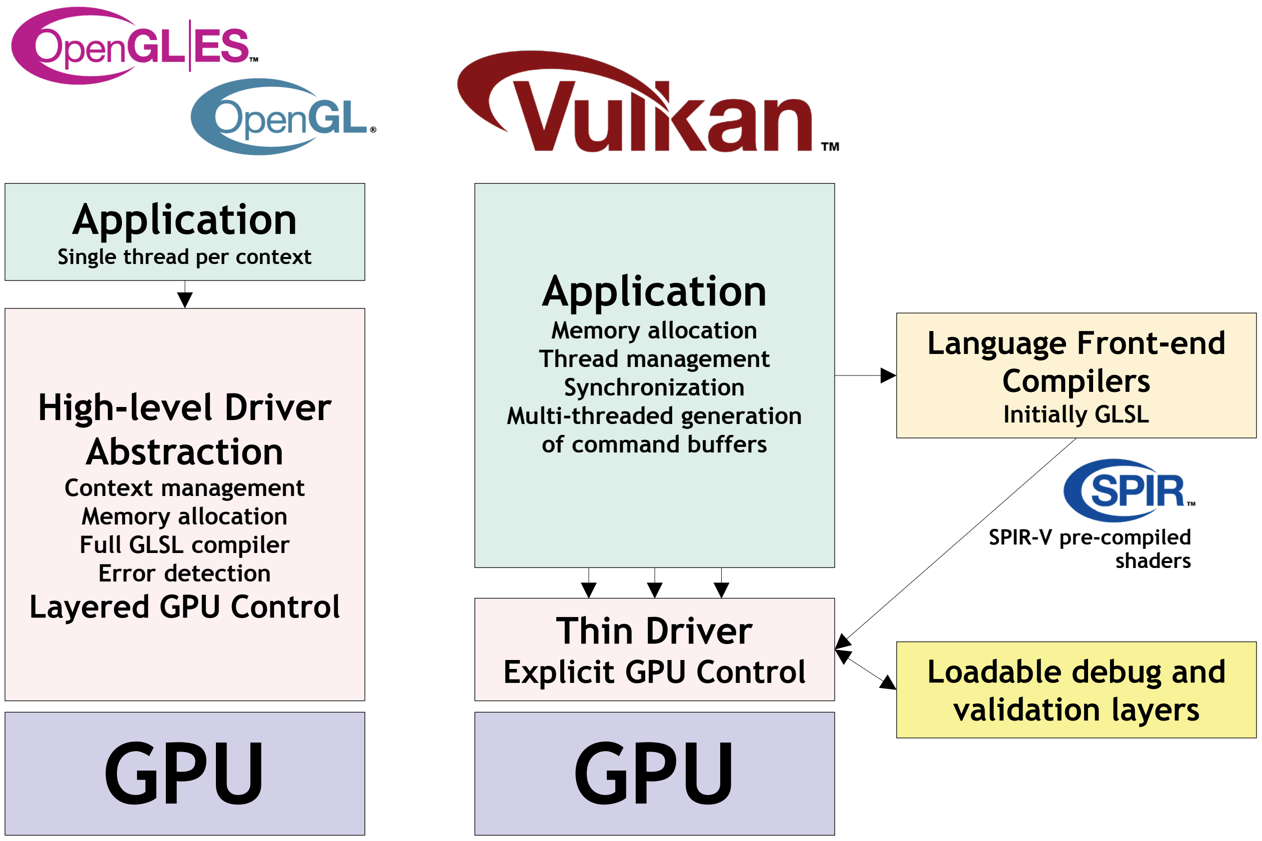
Only Windows 10

Only Apple

Cross Platform

Vulkan is extensible - just like OpenGL - and so is the new only generation API where hardware vendors can deliver innovations to market whenever they need

Vulkan Explicit GPU Control



Vulkan 1.0 provides access to
OpenGL ES 3.1 / OpenGL 4.X-class GPU functionality
but with increased performance and flexibility

Vulkan Benefits

Resource management in app code:
Less driver hitches and surprises

Simpler drivers:
Improved efficiency/performance
Reduced CPU bottlenecks
Lower latency
Increased portability

Multi-threaded Command Buffers:
Command creation can be multi-threaded
Multiple CPU cores increase performance

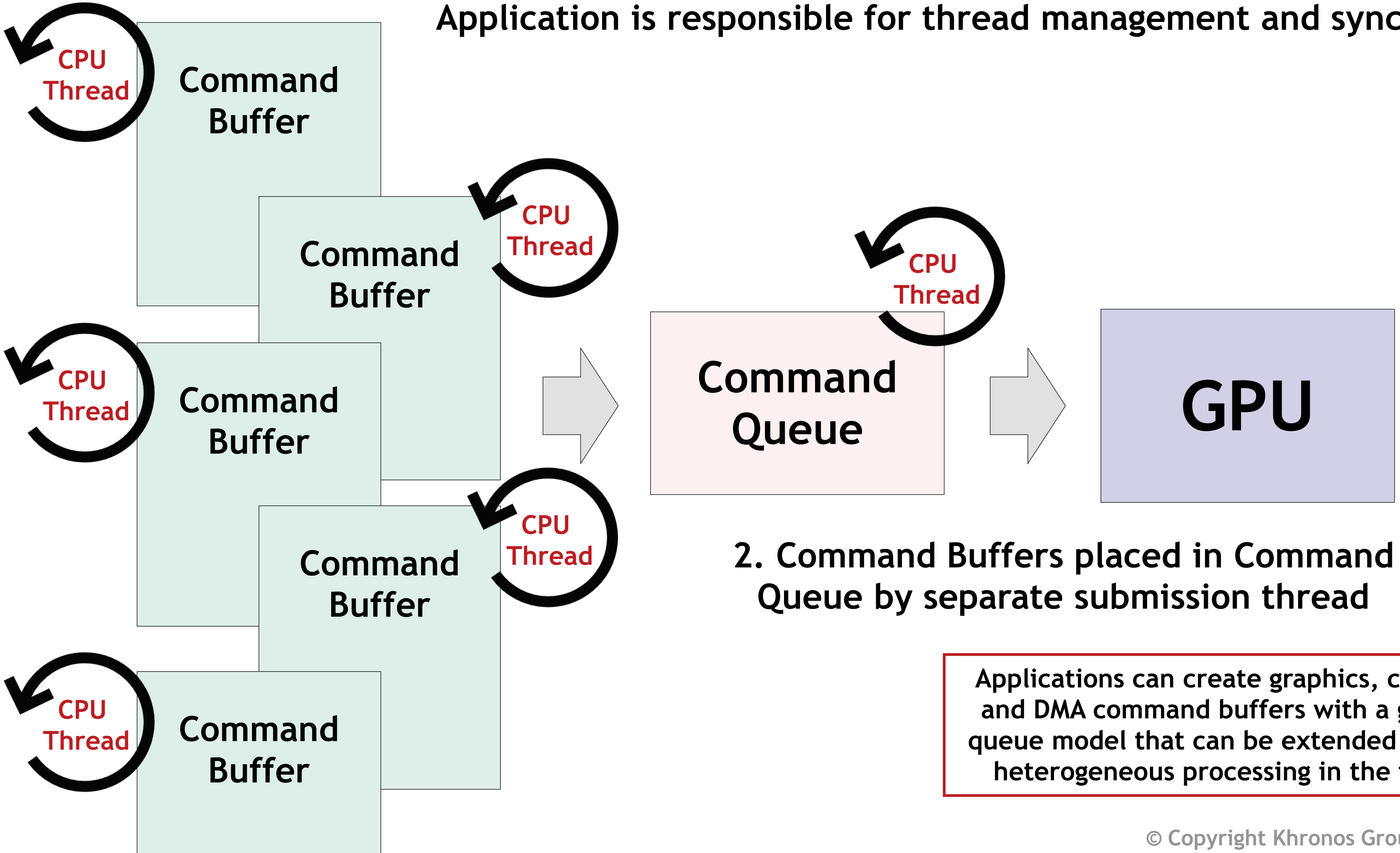
Graphics, compute and DMA queues:
Work dispatch flexibility

SPIR-V Pre-compiled Shaders:
No front-end compiler in driver
Future shading language flexibility

Loadable Layers
No error handling overhead in production code

Vulkan Multi-threading Efficiency

1. Multiple threads can construct Command Buffers in parallel
Application is responsible for thread management and synch



2. Command Buffers placed in Command Queue by separate submission thread

Applications can create graphics, compute and DMA command buffers with a general queue model that can be extended to more heterogeneous processing in the future

Vulkan Genesis

Khronos members from all segments of the graphics industry agree the need for new generation cross-platform GPU API

Unprecedented level of participation from game engine developers

Significant proposals, IP contributions and engineering effort from many working group members

18 months

A high-energy working group effort

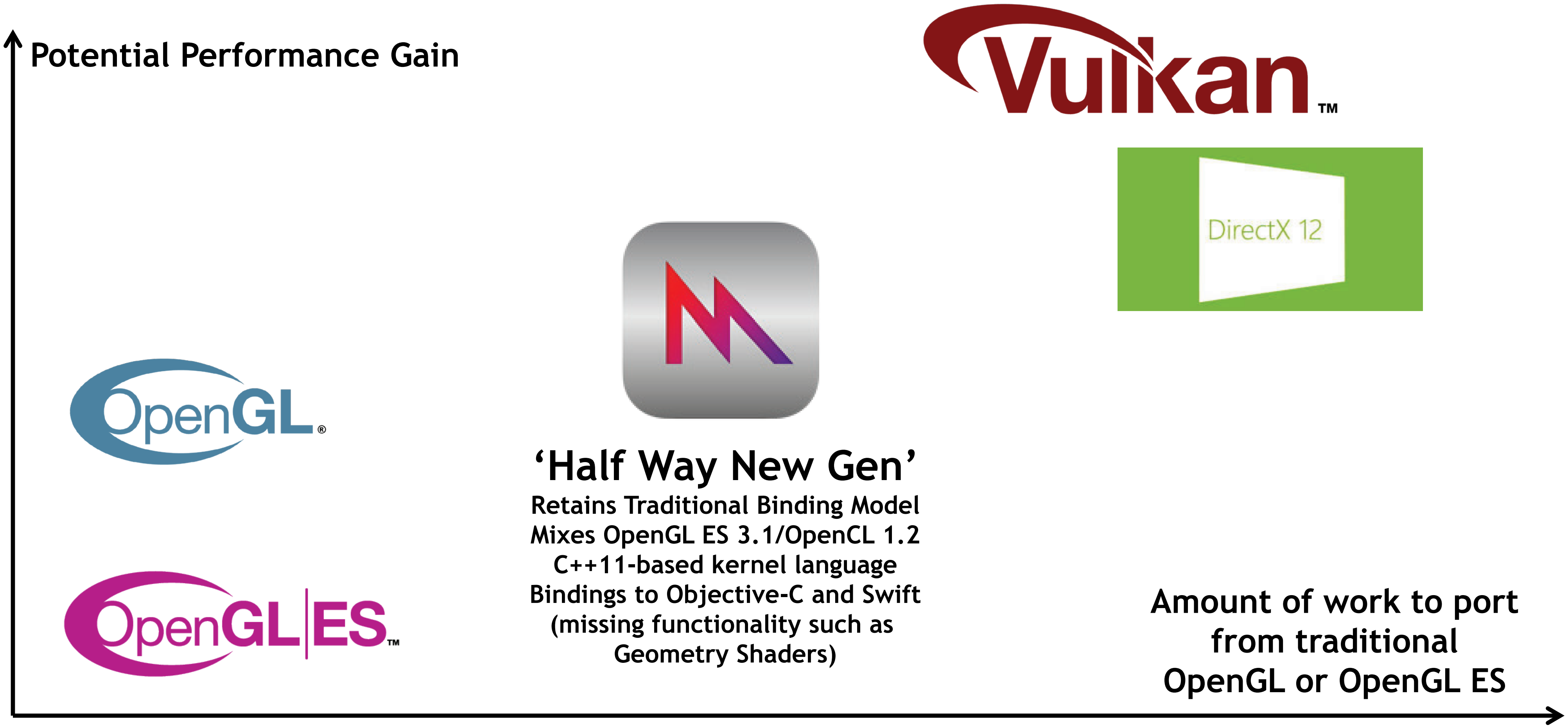
Vulkan

Khronos' first API 'hard launch' February 2016

Specifications, Conformance Tests, SDKs, Reference Materials, Compiler front-ends, Samples - all open source…
Multiple Conformant Drivers on multiple OS

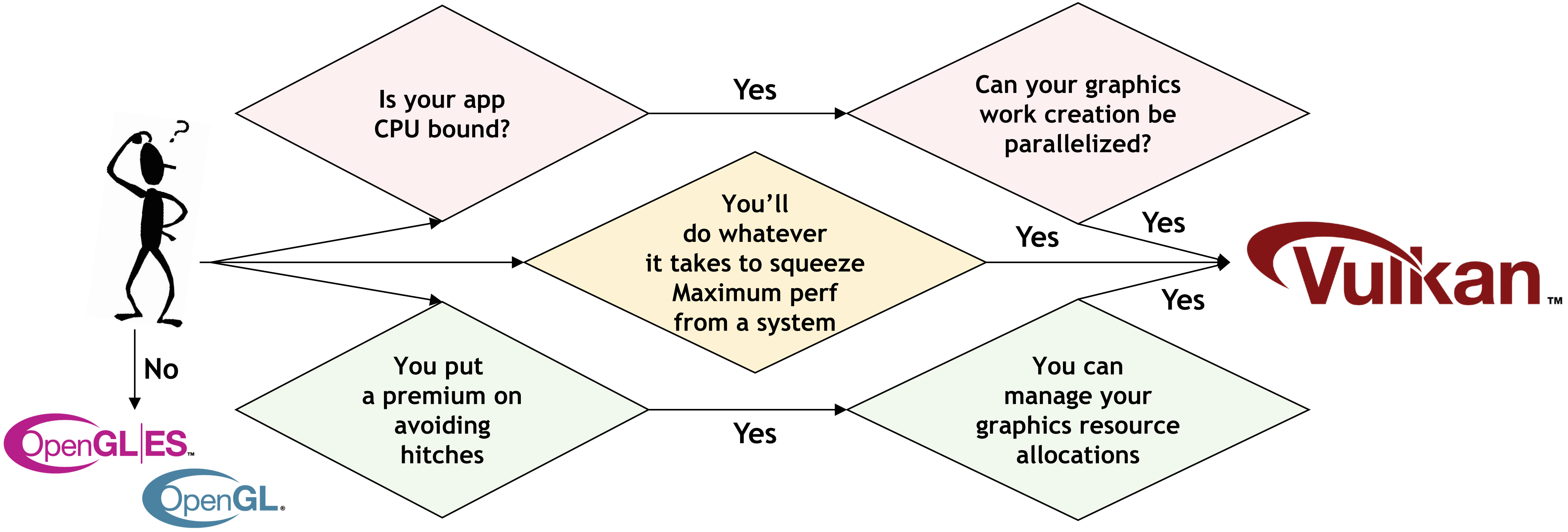
Vulkan Working Group Participants

Vulkan - No Compromise Performance



Which Developers Should Use Vulkan?

- **Vulkan puts more work and responsibility into the application**
  - Not every developer will need or want to make that extra investment
- **Vulkan provides a choice for developers**
  - For many developers OpenGL and OpenGL ES will remain the most effective API
  - Vulkan can be used to create new classes of end-user experience



The Power of a Three Layer Ecosystem

Applications can use Vulkan directly for maximum flexibility and control

Application uses utility libraries to speed development

Utility libraries and layers

Application

Game Engines fully optimized over Vulkan

Applications using game engines will automatically benefit from Vulkan's enhanced performance

Rich Area for Innovation

- Many utilities and layers will be in open source
- Layers to ease transition from OpenGL
- Layers to address intersection of Vulkan, DX12, Metal
- Domain specific flexibility
- Performance across diverse hardware

Similar ecosystem dynamic as WebGL

A widely pervasive, powerful, flexible foundation layer enables diverse middleware tools and libraries

SPIR-V Transforms the Language Ecosystem

- **First multi-API, intermediate language for parallel compute and graphics**
  - Natively represents structures in shader and kernel languages
  - https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf

Multiple Developer Advantages

Use same front-end compiler for all platforms
Ship SPIR-V - not shader source code
Simpler and more reliable drivers
Reduces runtime kernel compilation time

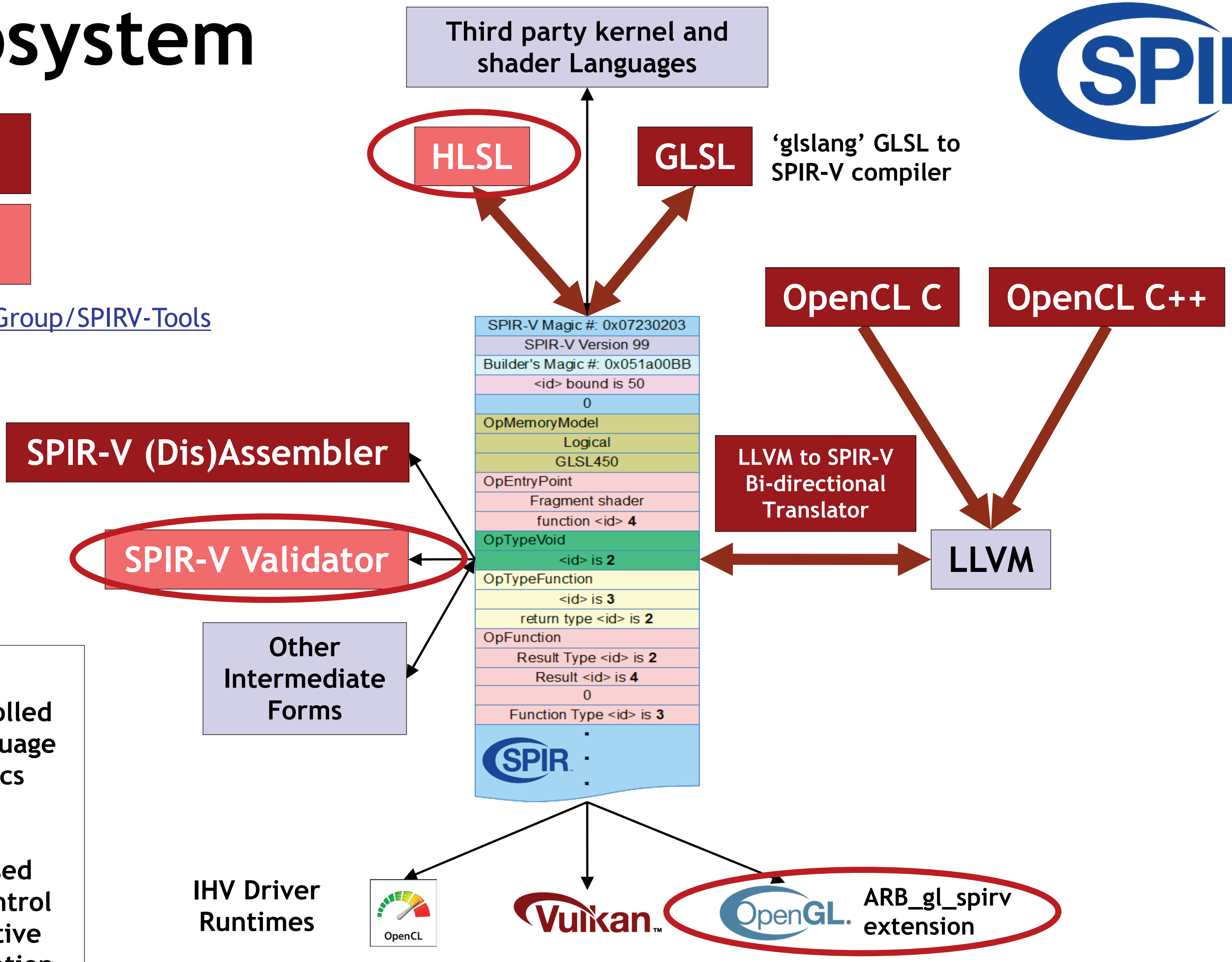
Standard
Portable
Intermediate
Representation

SPIR-V Ecosystem

Khronos has open sourced these tools and translators

Khronos plans to open source these tools soon

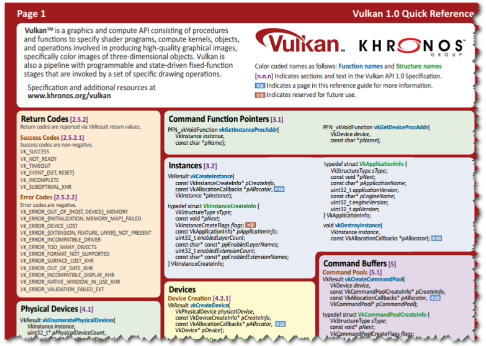
https://github.com/KhronosGroup/SPIRV-Tools



SPIR-V

- Khronos defined and controlled cross-API intermediate language
  - Native support for graphics and parallel constructs
  - 32-bit Word Stream
- Extensible and easily parsed
- Retains data object and control flow information for effective code generation and translation

Vulkan Developer Resources



KHRONOS GROUP

www.khronos.org/vulkan/

Canonical Resources

Specifications, Header Files
Feature Set Definitions
(Windows and Linux - post developer feedback)
Quick Reference and Reference Pages
Conformance Test Source and Test Process

Materials to Build SDKs and Tools

Compiler toolchain sources
Validation Layer Source
Loader Source
Layers and Loader documentation
(open source resources in github.com/KhronosGroup)

Everything needed to create SDKs for any platform or market

LunarG

Windows and Linux Installable SDKs
Loader and Validation Layer binaries
Tools Layers - source and binaries
Samples - source and binaries
Windows get started guide

IHV Websites

Drivers and Loader
Vendor tools and layers

Third Party Websites

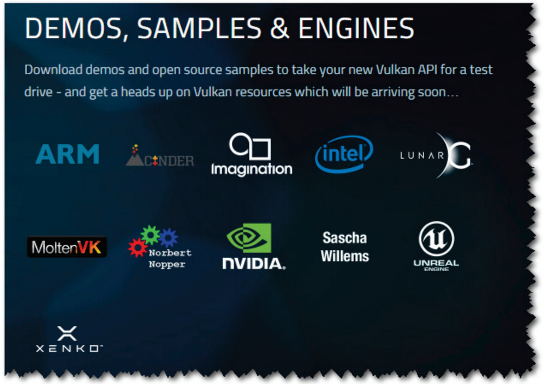
Layers, Samples etc.

LunarG® Vulkan™ SDK

As the first comprehensive Vulkan SDK for Windows® and Linux operating systems, the LunarG® Vulkan™ SDK includes everything you need to get started in the Vulkan API development environment.

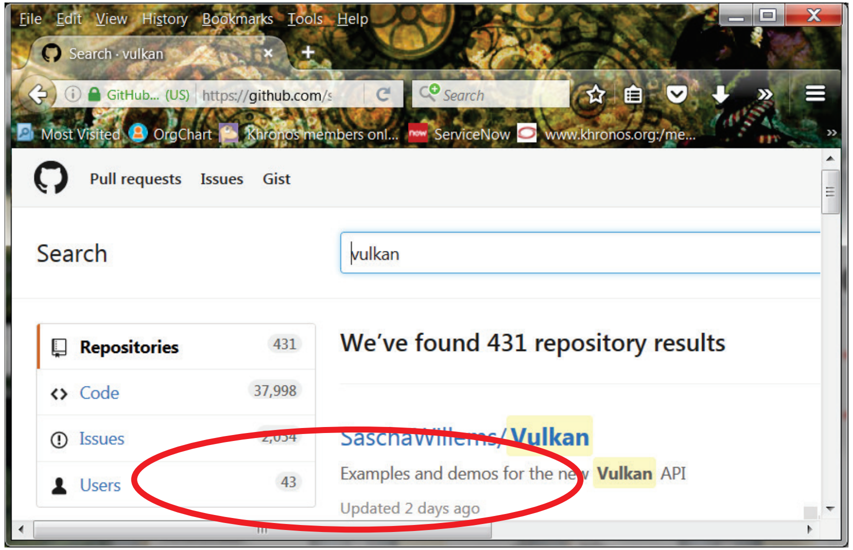
Download the LunarG Vulkan SDK for Windows or Linux
The SDK is open-source and freely available to all.
DOWNLOAD

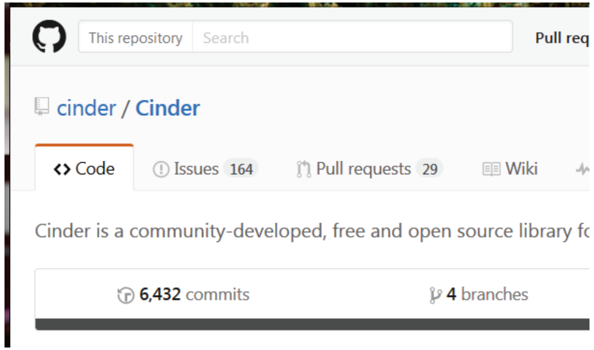
https://lunarg.com/vulkan-sdk/



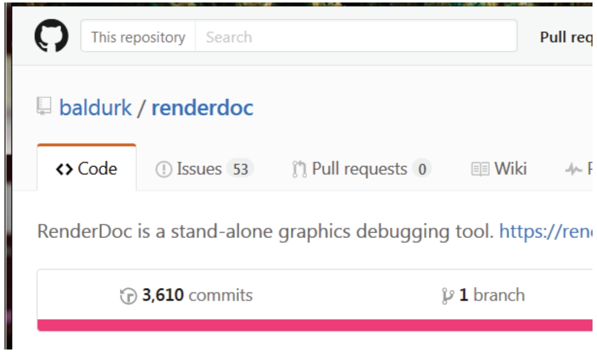
Vulkan Open Source

- A huge amount of Vulkan activity on GitHub!
  - https://github.com/KhronosGroup
  - Please get involved! All projects under Apache 2.0
- Open Source Tools
  - RenderDoc - Graphics debugger
    - Baldur Karlsson
  - Vulkan-hpp – C++ Wrapper for Vulkan
    - Markus Tavenrath / Andreas Süßenbach, NVIDIA
  - SPIRV-Cross - Cross-compiler / reflection tool
    - Hans-Kristian Arntzen, ARM

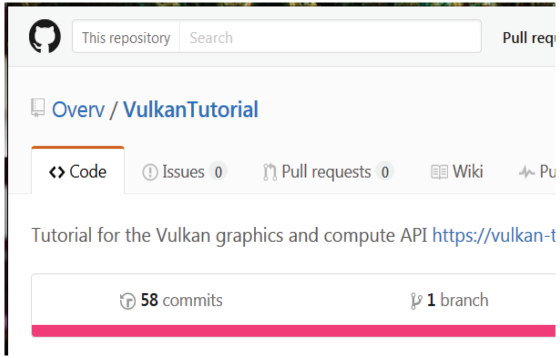




Ports



Tools



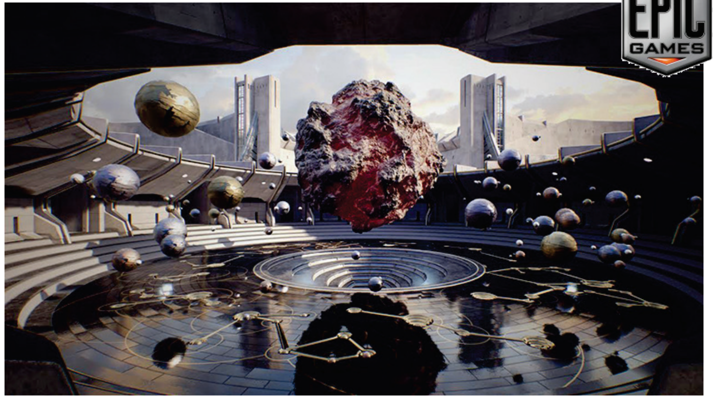
Tutorials

Vulkan Adoption- Hardware

- Conformant GPUs
  - AMD, ARM, Imagination, intel, NVIDIA, Qualcomm, VeriSilicon
- Desktop Hardware
  - AMD GCN (production)
  - Intel Skylake and Broadwell (beta, production coming soon)
  - NVIDIA GeForce and Quadro boards on Windows and Linux (production)
    - Kepler, Maxwell, Pascal GPUs
- Mobile and Embedded Hardware
  - Samsung Galaxy S7
  - NVIDIA Shield Tablet, Shield Android TV, Jetson TX1 (Linux)
  - Google Nexus 5X, 6P, Player, Pixel C (Android N Developer Preview)
  - *Lots* more on the way!

Vulkan Adoption - Games and Engines

Dota 2 on Vulkan port of Source 2

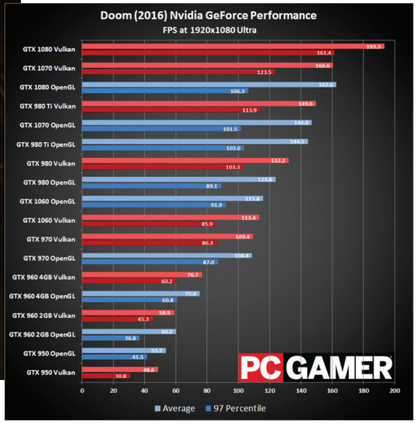


'ProtoStar' demo on Vulkan port of Unreal Engine 4

Talos Principle on Vulkan port of Serious Engine

Doom's Vulkan patch is a PC performance game-changer

DOOM on Vulkan port of id Tech 6



CRYENGINE CRYTEK

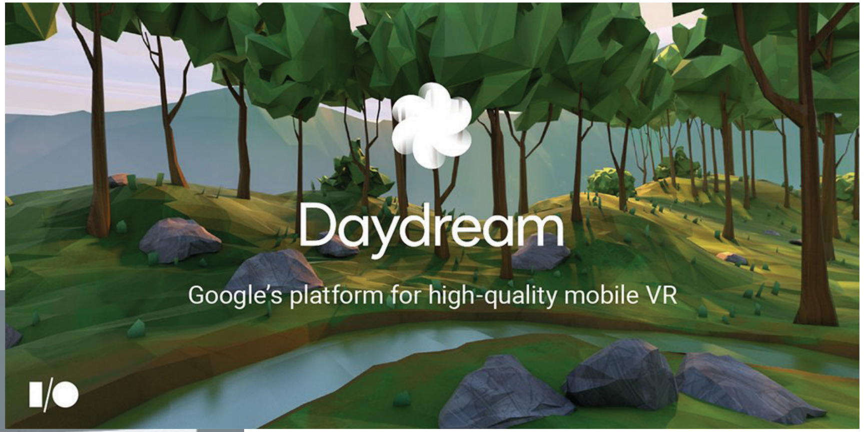
Vulkan support in Mid October 2016

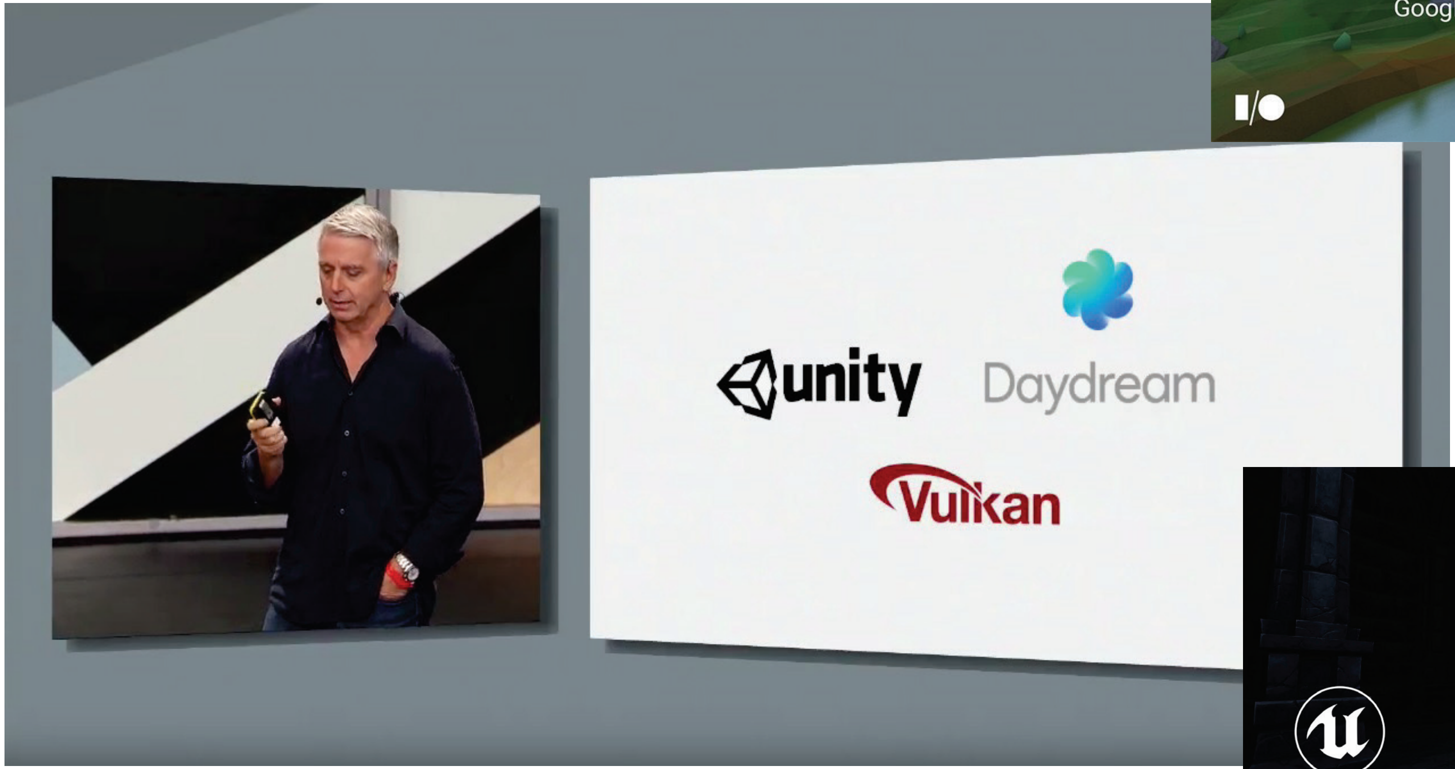
XENKO Silicon Studio

Vulkan support in V1.8

Android Daydream VR

Daydream uses Unity and Unreal
- accelerated over Vulkan





MoltenVK

- **MoltenVK is an implementation of Vulkan on iOS & MacOS**
  - Made by Brenwill Workshop
  - Built over Metal
- **Vulkan & Metal are both static-state, command-buffer APIs**
  - Small overhead - so MoltenVK has good performance
- **MoltenVK feature set dependent on Metal**
  - Metal's focus is on providing a convenient API
  - MoltenVK is a subset of Vulkan - helps define cross-platform compatibility

https://moltengl.com/moltenvk/

Vulkan application running on Apple iPad

Khronos Roadmap Possibilities in Discussion

SPIR-V Ingestion in OpenVX and OpenGL for programmable node and shading language flexibility

Vulkan Roadmap Priorities

- Multi-GPU
- Virtual Reality (asynch compositing, efficient multi-view rendering, direct screen access)
- Cross-API/process resource/event interop
- Subgroup instructions (e.g. shader ballot)
- Rigorous memory model

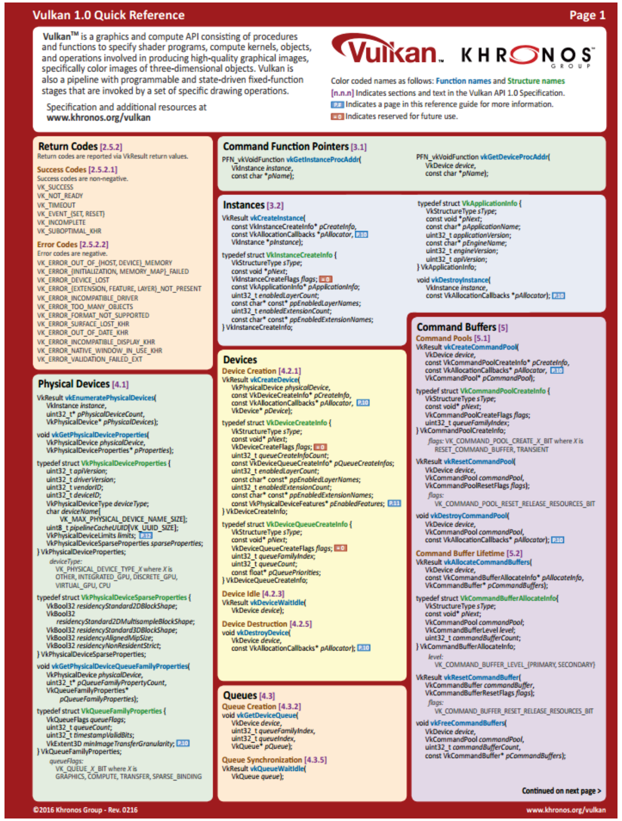
Thin and predictable graphics and compute for safety critical systems

1. C++ Shading Language
2. Single source C++ Programming from SYCL
3. OpenCL-class Heterogeneous Compute to Vulkan runtime

Khronos members decide how to evolve and mix and match a rich set of APIs and technologies to meet market needs

Thank You!

- **Please contribute to the Vulkan ecosystem**
  - Give us feedback!
- **NVIDIA Vulkan developer hub**
  - www.khronos.org/vulkan/
  - We need examples, tutorials, demos, tools…
  - *Note - watch for RFQs forthcoming at www.khronos.org*
- **Let us help you promote your use of Vulkan!**
  - Got a cool Vulkan-generated video?
    Let us host and promote it!
  - Send mail to marketing@khronos.org
- **Any company or organization is welcome to join Khronos for a voice and a vote in the evolution of Vulkan!**
  - www.khronos.org



https://www.khronos.org/files/vulkan10-reference-guide.pdf

Jungwoo Kim, Samsung mobile

Advanced Mobile Gaming with Vulkan

SAMSUNG mobile

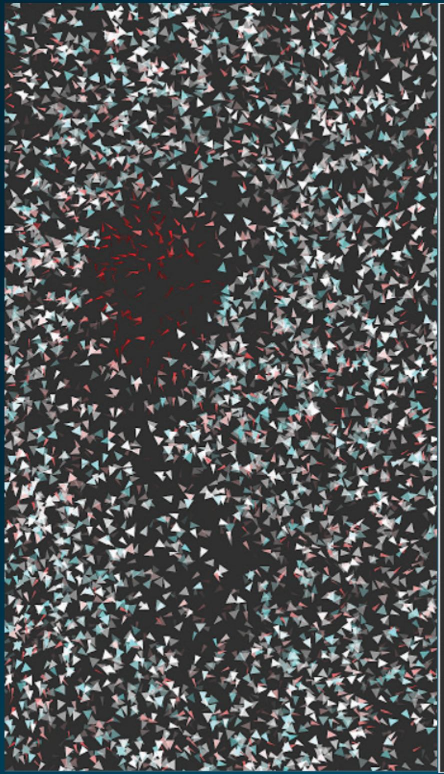
We did several things about Vulkan

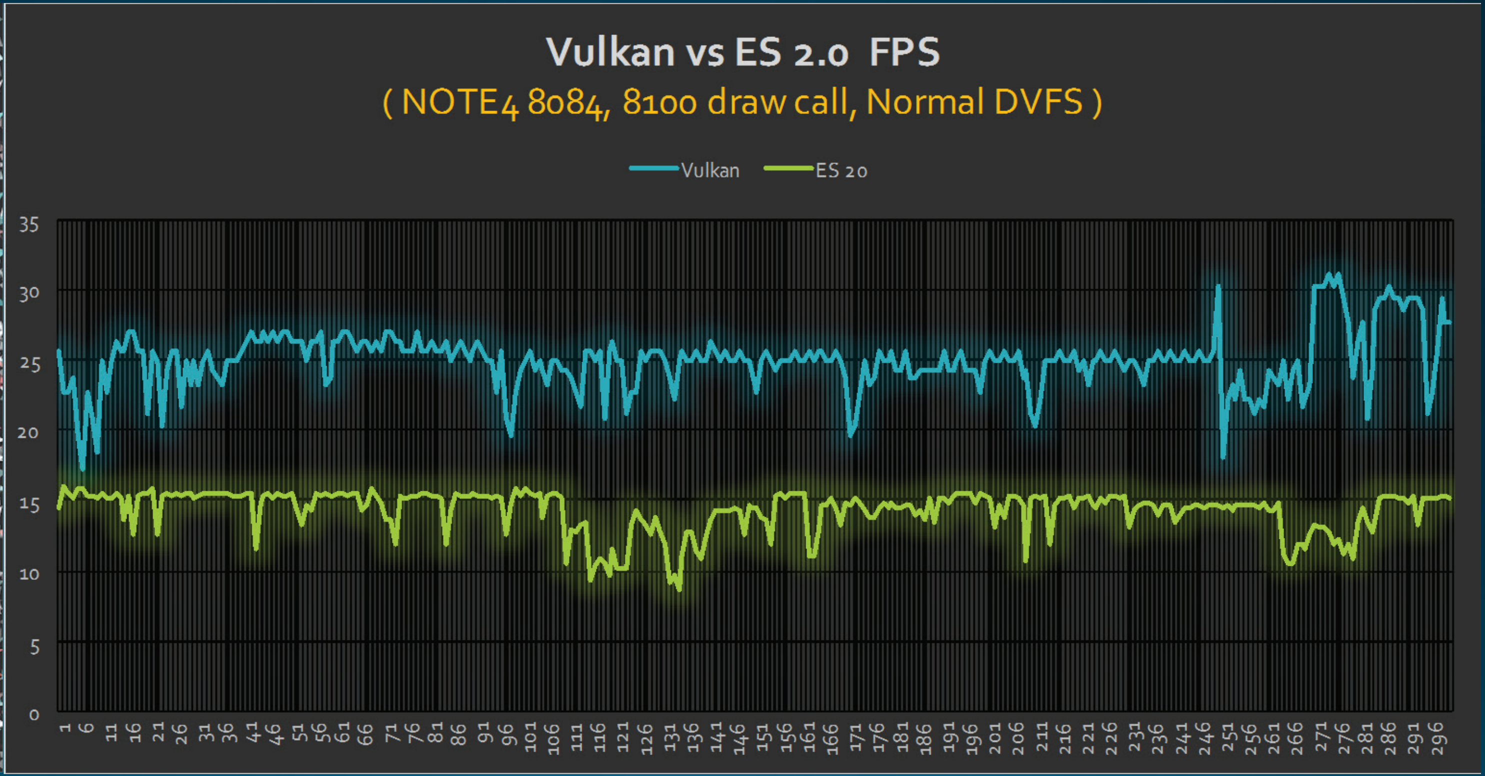
- Early studies with corner case examples
- Proof of concept level demo implementation
- Do collaboration for porting real game engine
- Do collaboration for developing cool Vulkan demo
- Developing real Vulkan games for market

Exciting 1 year for Vulkan with Khronos and great game companies!

SAMSUNG mobile

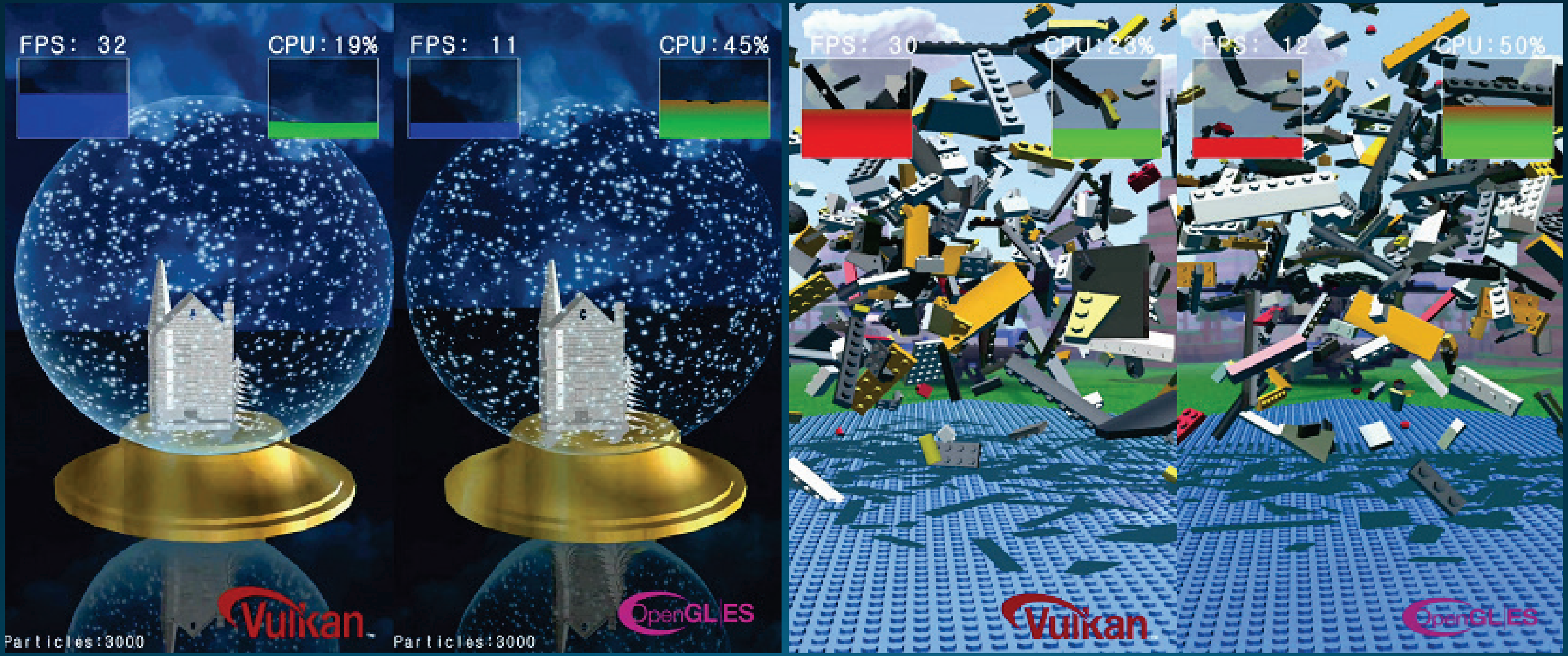
Early studies : Heavy Drawcalls!!!





AVERAGE 23.2 FPS / 14.5 FPS

Proof of Concept demo



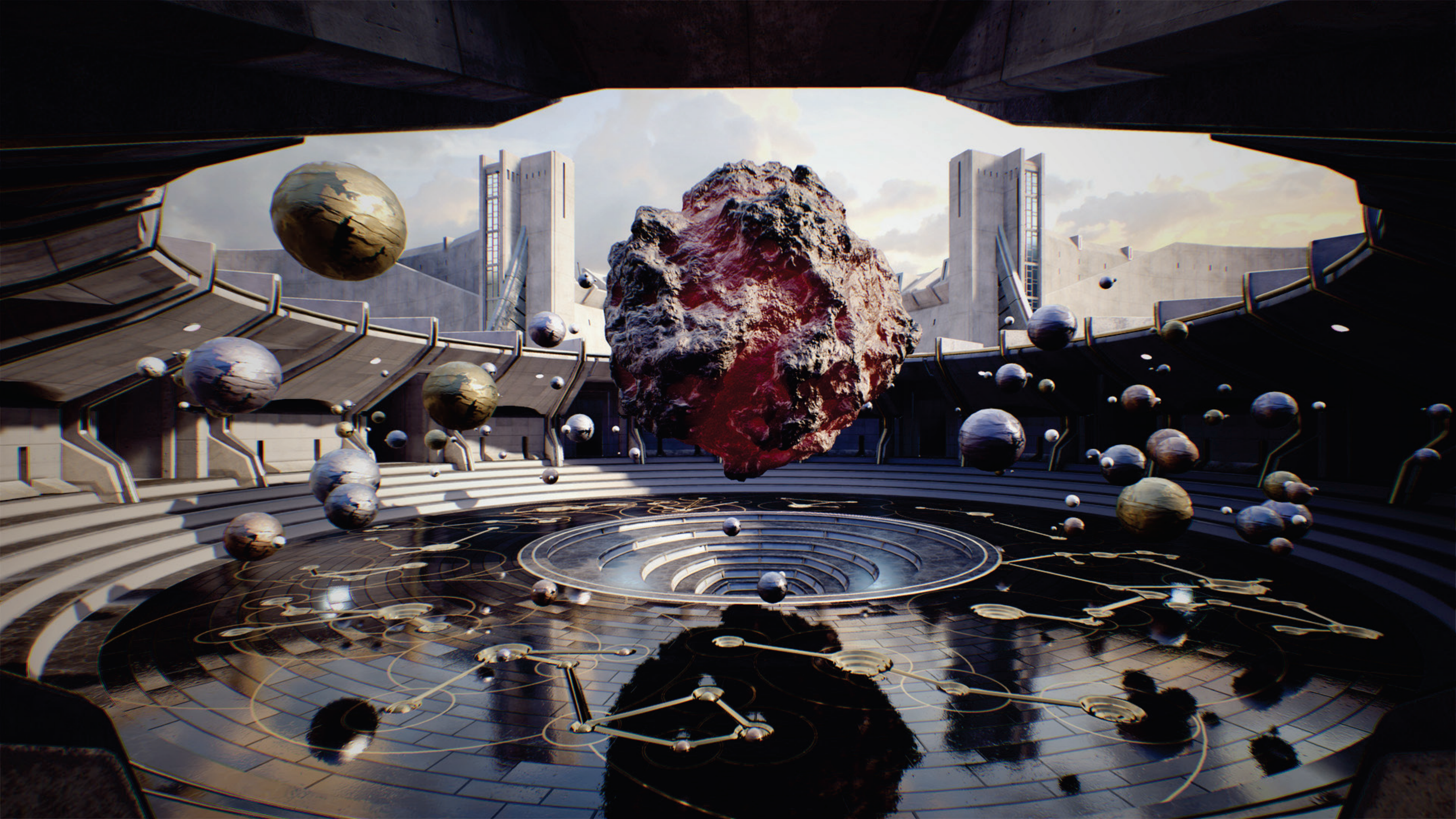
SAMSUNG
mobile

Porting Real Game Engine

- Ronin project with Epic Game for porting UE4 support Vulkan
- First goal was running Tappy chicken and SunTemple

SAMSUNG
mobile

Developing cool demo : ProtoStar



Developing Real Vulkan Games

- We decided support game companies to port their games
- Tight schedule pushed us to focus on some specific directions

SAMSUNG mobile

VainGlory : Performance Gain

- VainGlory was already well optimized and still near 60 fps in GLES
- But Vulkan gives everlasting 60 fps in any case with huge reduction in memory usage by ASTC

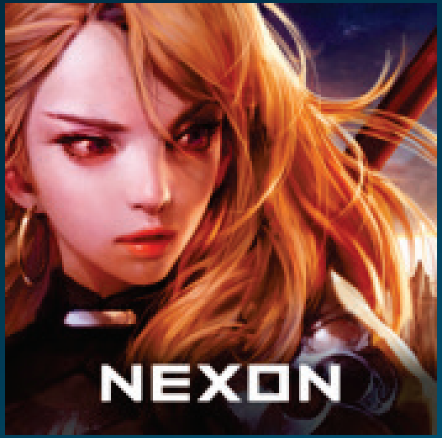
Performance	Normal	4 %
	Throttling	30 %
Power usage		5 %
Memory usage		25 %

SAMSUNG mobile

34
13:52
e302
12
e339
12
You have unspent ability points
Frame Rate Stability
OpenGL|ES
1e6-86d6-12313f0484e2 Connected at: dev-ext.superevilmegacorp.net, plgt_state: Playing, client_state: PLAYIN
54
13:52
e302
12
e339
12
You have unspent ability points
Frame Rate Stability
Vulkan
11e6-9b39-12313f0484e2 Connected at: dev-ext.superevilmegacorp.net, plgt_state: Playing, client_state: PLAY

HIT : More Visual Quality

- HIT is UE4 based mobile action RPG game
- We focus on adding more graphics effects with same performance



SAMSUNG mobile


mosh53
9,999,999/9,999,999
1
00:27
samsung-0.65046.67662
52.196.22.59:8000
20 Lv
10 Lv
자동
전투

HIT : More Visual Quality

- Samsung back ported all Vulkan RHI and new features related changes from Ronin and UE4.12 into old version of UE4
- Advanced reflection, refraction, GPU particles and cinematic depth of field effect were added
- NATGames changed stage design, assets and camera view to maximize the impact of all these new graphics effects
- After adding and changing things, performance is exactly same and game build passed publisher's QA process with both chipsets
- Spending 6 weeks for E3 demo and another 6 weeks for market

SAMSUNG mobile

Wrap-up

- Heavy drawcall is not only case having benefit of Vulkan in mobile
- Vulkan gives CPU off-load, predictable behavior by explicit control and various ways to optimize games
- Vulkan drivers are not perfect but stable enough to make a game
- ASTC and SPV is essential, Vulkan will give more power with them
- No more driver magic, so you need to manage things by yourself

We will keep go on contributing to Vulkan and gaming industry!

SAMSUNG mobile

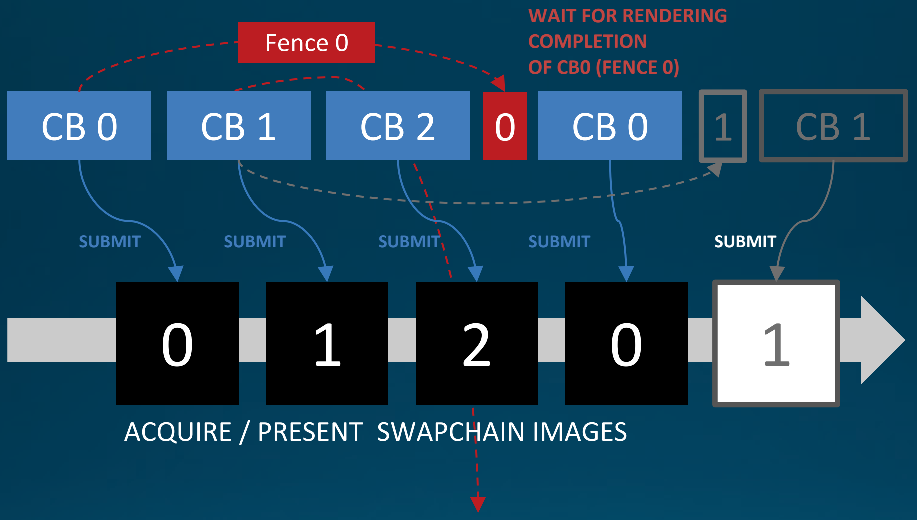
Thank you!

If you have any questions, please contact

jwoo.kim@samsung.com or gamedev@samsung.com

SAMSUNG mobile

Back-ups : Vulkan Tips 1/3



- Recommend to use A big primary command buffer for corresponding back-buffer and need to try to submit once a frame *(e.g. 3 back buffers requires at least 3 command buffers)*
- Should use "FIFO" instead of "MAILBOX" for SwapChain creation on Android to get stable Queue/Dequeue based on vsync

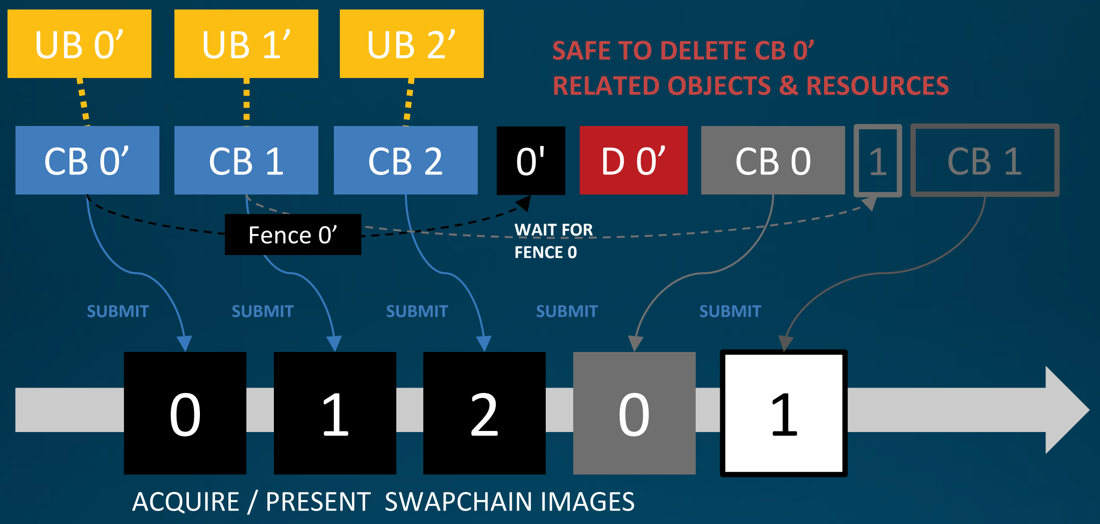
SAMSUNG mobile

Back-ups : Vulkan Tips 2/3

- Recommend to not to bind render state every draw calls, need to remove duplications.
  - *there will be cases vkCmdSetXXX, vkCmdBindXXX functions are already bound at command buffer.*
- Recommend to use pipeline cache and caching everything as much as you can.
- Efficient managing of vkGraphicsPipeline is key point of rendering optimization. (e.g. using map)
- Recommend to not to use local fence usage. (local blocking command buffer submit)
- Recommend to use proper image layout for image resources. Especially for presentation and copy.
- Recommend to use fixed **UniformBuffer** for static objects to reduce vkUpdateDescriptorSets calls.
  - *and also fixed uniformbuffer can be used for secondary command buffer optimization*
- Recommend to use **UniformBufferPool** for dynamic objects
  - *Calling vkMapMemory, vkUnmapMemory can be optimized by mapping one single big memory and referencing this memory through the several uniform buffers*
  - *But, Please keep mind that should consider minMemoryMapAlignment for memory offset.*

SAMSUNG mobile

Back-ups : Vulkan Tips 3/3



- Should use deferred deletion for vulkan related resources & objects
  - *- Recommend to wait 3 frames till delete candidates are safely detached from it’s command buffer.*
- Recommend to use at least same count of **UniformBuffers** for it’s **CommandBuffers**.
- Please keep in mind that uniformbuffers at shader side should follow **‘std140 layout’** rules.
- Should use **uvec3** instead of **ivec3** or **vec3**, when you using VK_FORMAT_R8G8B8_UINT for attributes type.
- Enabling validation layer will be very helpful to correct API usage but should not trust error result 100%

SAMSUNG mobile

Silicon Studio

Silicon Studio

Vulkan Implementation in Xenko

Presentation by Pierre Rahier
http://www.siliconstudio.co.jp/
pierre@siliconstudio.co.jp

XENKO™

What is Xenko?

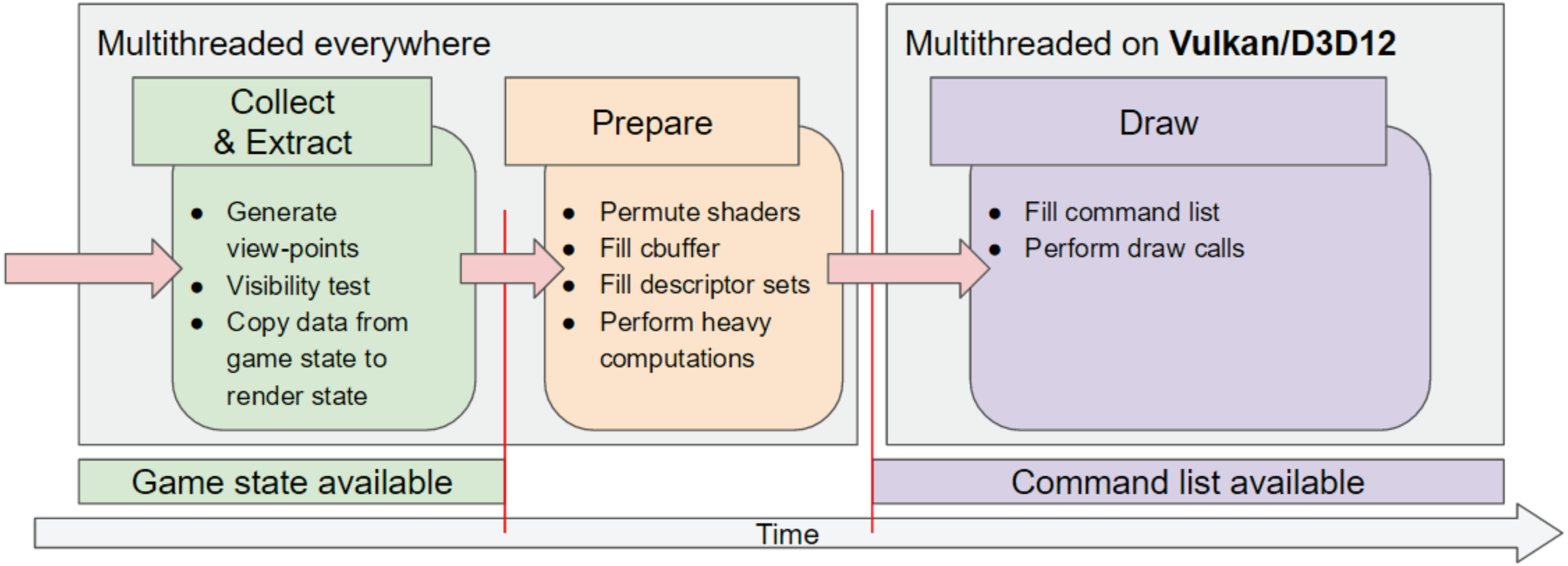
Next Generation Game Engine

- Developed by Silicon Studio
- Preparing for virtual reality revolution
- Open source, free Beta version available
- Fully integrated C# 6.0 engine
- Cross platform support (mobile, PC, console)
- Designed Xenko rendering pipeline for next-gen APIs from inception
    - Multi-threading the engine and porting to Vulkan helped the transition

Porting Xenko to Vulkan

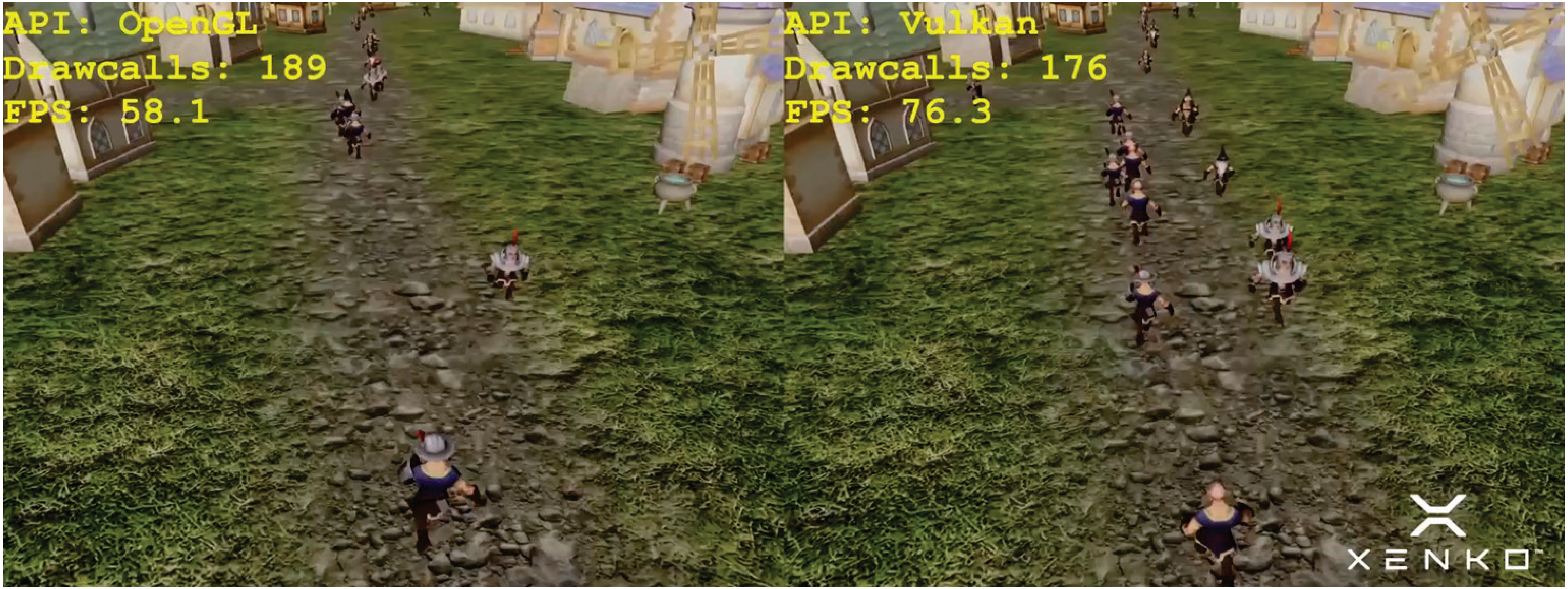
Designed our rendering pipeline for next-gen APIs

- Created our abstraction graphic layer (best of Vulkan & DirectX 12)
- Fine-grained parallelism for better scalability (instead of “system-on-a-thread”)



Results

- Benchmark demo (available on youtube)



※ Benchmarked on: Core i7 3.7 GHz, AMD Radeon RX 480

Results

- Same Benchmark
- 3.6Ghz, 4 cores CPU
- NVidia Driver
- CPU-Bound

API	Draw calls/second※	Speed Increase
OpenGL	772k	100%
DirectX 11	1164k	151%
Vulkan	959k	124%
Vulkan Multi-threaded	2840k	368%

※ *Numbers above are in draw calls per seconds of rendering*

Our Experience

Difficulties

- Varying driver behaviors & performances (NVidia / AMD)
- CPU/GPU sync points hard to identify

Advantages

- No overhead for API calls
- Designed for multi-threading
- Easier debugging than OpenGl, DX11, DX12

Future Work

- Optimize Vulkan implementation further
- Making full use of advanced Vulkan-features
- Adopt SPIR-V as internal shader preprocessing language

Thank you for listening!
ご静聴ありがとうございました!

For updates, follow us!

https://twitter.com/xenko3d

https://www.facebook.com/xenko3d/

DMPとVulkan

株式会社ディジタルメディアプロフェッショナル
大渕　栄作

Aug/2016

Graphics SoC: VF2

- 組み込み機器向け高性能グラフィックスLSI
  - 3Dグラフィックス機能(OpenGLES及び独自拡張)と2D画像コーディック機能を1チップに統合したSoC

LSI Overview	
CPU	ARM Cortex-A7 Quad
Memory	16Mbyte embedded VRAM DDR3-1600/DDR3L-1333
GPU	OpenGLES3.0 GPU Quad core OpenVG1.1 Vector graphics 2D blending acc.
Video	H.264 decoder 1920x1080 @ 30fps x 24
Bus fabric	DMP Loputo - Memory controller / Interconnect bus
Display	Full HD 2画面同時表示 (LCD controller x 5)
Fab	TSMC 28nm

上記LSI向けのVulkan環境を準備中
⇒ OpenGL ES 環境からVulkan環境への移行における基本情報をご紹介

OpenGL ESから Vulkanへの移行

OpenGLES⇒Vulkanに移行した方がいい理由

- ドライバ処理が重いアプリ
  - 元々GPUネックのアプリや、ドライバ以外CPU側の処理が重いのアプリは Vulkanを使ってもほぼ性能の向上を得られません。
  - Vulkanでは一度生成したコマンドバファは使いまわしが可能 ⇒ ドライバの処理時間を大幅に減らせる可能性
- Explicitな制御により、フレームレートなどを制御したい
  - OpenGLES はドライバ内部でリソース管理やshaderの隠しコンパイルの原因で違うフレームで処理の時間が大きく変動する場合がある。
  - Vulkanは全部アプリで管理になるので、重い処理を違うフレームに分散することで一定のフレームレートを実現
- マルチスレッドでGPUのコマンドを生成
  - OpenGLESはSingle Threadの環境下で動くの設計⇒Vulkanは設計からマルチスレッドでコマンド生成とリソースシェアを考慮しています。
  - すべてのオブジェクトは全部のスレッドに共有できますので、複数のCPUコアを同時に使ってGPUコマンドの生成は可能です。

OpenGLES/EGL⇒Vulkan APIのマッピング

- OpenGLとVulkan設計上の違い
  - OpenGL(ES)はstate machine、全部の操作は順番通り処理
  - 2X年前から当時の設計をずっと使ってきたが、現在のHWやrendering pipeline(特にdeferred render)とマッチしていない。
  - ドライバは処理を加速するため、内部でいろいろの隠し処理 (e.g. メモリの管理、非同期のための臨時バファ)
- Vulkanがおこなう事
  - GPUのリソースを最大限に有効利用するため、元々ドライバがやっていた隠し処理を全部アプリにまかせて、アプリは本当に必要な処理だけを発行でき、無駄な処理を削れる
  - OpenGL(ES)と比べて、Vulkanはコマンドバファ、コマンドキュー、メモリ管理、バファの更新とかのAPIを追加し、それらの操作もアプリになる⇒手間は増えるが、OpenGLよりも高速に動作する可能性がある

DMP DIGITAL MEDIA PROFESSIONALS

既存アプリのポーティング

- 既存アプリをそのままポーティングでも効率は上がらないかも
  - VulkanとOpenGL(ES)の設計は違うので、既存のアプリのGLES APIを対応のVulkan APIにそのまま替えても（いわゆるべた移植）効率の改善は見られないかも
- 完全にVulkanの利点を引き出すには、根本からVulkanの設計に沿ったアプリを作り直す必要があり

OpenGLES (EGL) ⇒ Vulkanのマッピング

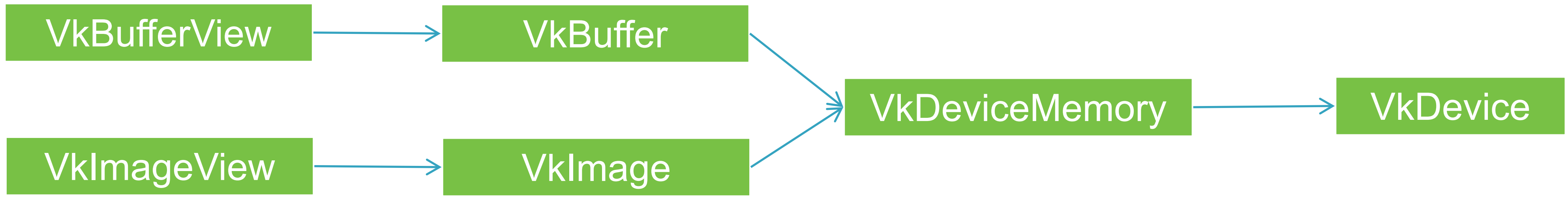
- Program/Shader
  - Program ⇒ VkPipeline
  - Shader ⇒ VkShaderModule
  - ProgramPipeline⇒VkPipeline
  - (Resource binding) ⇒ VkPipelineLayout, VkDescriptorSet
- Buffer, Image, Sampler
  - Buffer ⇒ VkBuffer
  - UniformBuffer ⇒ VkBufferView
  - SSBO ⇒ VkBufferView
  - Framebuffer ⇒ VkFramebuffer, VkRenderPass
  - Renderbuffer ⇒ VkImage, VkImageLayout
  - Texture ⇒ VkImage, VkImageLayout
  - Image ⇒ VkImageView
  - Sampler ⇒ VkSampler
- Other
  - TransformFeedback ⇒ None (TF is removed from Vulkan)
  - VertexArray ⇒ VkPipeline
  - Querie ⇒ VkQueryPool
  - (Other States) ⇒VkPipeline
  - Sync ⇒ VkFence
  - (Command Buffer) ⇒ VkCommandBuffer
  - (Render Queue) ⇒ VkQueue
- EGL
  - EGLDisplay ⇒ VkPhysicalDevice, VkDevice
  - EGLContext ⇒ VkQueue, VkCommandBuffer
  - EGLSurface ⇒ VkSurfaceKHR, VkSwapchainKHR

完全に1:1マッピングではないが、概念的に近いモノ

デバイスメモリ

- Heap, Memory, Resource, View
  - Heap: Device上存在するメモリ。属性によって数種類（e.g. CPUでmmapできるメモリ領域、Device上の速いメモリ領域とか。）
  - Memory: DeviceからVkAllocateMemory()でメモリを取得。そのメモリをresourceにattachして、該当のresourceのメモリとして使用
  - Resource: Vulkanでは主に二種類のresource（BufferとImage）
    - Bufferはvertex data やuniform dataが入る。
    - Imageはtextureあるいはrendering targetになる。
    - Resource生成時はメモリが配置されない。
    - vkBindBufferMemoryやvkBindImageMemoryでそれぞれ使うメモリをattachする。
  - View:
    - Buffer viewはshaderからbufferをaccess する時の、オフセットとフォーマットの情報
    - Image viewは shader から image を access する時のフォーマットとrangeの情報
-

Vulkan API object関係図



Resource management

- メモリ配置は重い処理なので、アプリは最初に大きめなメモリブロックを取得し、各リソースに自分でsub-blockに分割する実装を推奨
- リソースをコマンドバファに送る前に関連のメモリをattachする必要があり、該当のコマンドが完了まで関連のメモリは解放できない
- これら処理はVulkanではアプリの責任になっている。

Memory Aliasing

- OpenGLESと違って、Vulkanのbufferとimageは自分でメモリを管理できる
  - 違うresourceの間で同じメモリ区間をattachするとそのメモリ区間はattachされたリソースに共有されます。

    ⇒Memory Aliasingと呼ばれる
- メモリの共有は任意のリソース間で可能
  - BufferとBufferやImageとImageだけではなく、BufferとImageも可能
  - 生存期間がoverlapしていないリソースはメモリの共有により使用メモリを節約
- 共有されたリソース間では一つのリソースに対しての書き込みは他のリソースも自動的に反映される
  - 注意として　ImageはtilingがVK_IMAGE_TILING_LINEAR、かつlayoutがVK_IMAGE_LAYOUT_PREINITIALIZEDあるいはVK_IMAGE_LAYOUT_GENERALの時だけ、CPUからそのメモリに対してのaccessが有効になっている。
  - BufferとImageの間にメモリの共有をする時、そのImageがCPUからaccessが有効じゃないと、Bufferの書き込みの結果はImage側からみるとUndefinedになる。

Vulkan上での実装例 (SDKサンプル)

初期化からコマンドキューとコマンドバファ生成

```
  // Instanceを取得します
  err = vkCreateInstance(&inst_info, NULL, &demo->inst);
  // physical deviceをリストを取得します
  err = vkEnumeratePhysicalDevices(demo->inst, &gpu_count,
physical_devices);
  // 先頭のphysical deviceを使う
  demo->gpu = physical_devices[0];

  // Deviceを生成
  err = vkCreateDevice(demo->gpu, &device_info, NULL, &demo->device);
  // Queue propertiesを取得します
  vkGetPhysicalDeviceQueueFamilyProperties(demo->gpu, &demo-
>queue_count, demo->queue_props);
  // GRAPHICSの描画をサポートするqueueを探します
  for (i = 0; i < demo->queue_count; i++)
    if ((demo->queue_props[i].queueFlags &
VK_QUEUE_GRAPHICS_BIT) != 0)
  graphicsQueueNodeIndex = i;
  //描画のqueueを取得します
  vkGetDeviceQueue(demo->device, demo->graphics_queue_node_index,
0, &demo->queue);

  // Command Poolを生成
  err = vkCreateCommandPool(demo->device, &cmd_pool_info, NULL,
&demo->cmd_pool);

  // command buffer を生成します
  vkAllocateCommandBuffers(demo->device, &cmd_info, &demo->cmd);
  // VkSurfaceKHRを生成します
  // Win32
  err = vkCreateWin32SurfaceKHR(demo->inst, &createInfo, NULL, &demo-
>surface);
  // Android
  err = vkCreateAndroidSurfaceKHR(demo->inst, &createInfo, NULL,
&demo->surface);

  // Swapchainを生成します
  const VkSwapchainCreateInfoKHR swapchain = {
    .sType =
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR,
    .pNext = NULL,
    .surface = demo->surface,                                  //
VkSurfaceKHRをいれます
    .minImageCount = desiredNumberOfSwapchainImages,
    .imageFormat = demo->format,
    .imageColorSpace = demo->color_space,
    .imageExtent =
      {
       .width = swapchainExtent.width, .height = swapchainExtent.height,
      },
    .imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT,
    .preTransform = preTransform,
    .compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR,
    .imageArrayLayers = 1,
    .imageSharingMode = VK_SHARING_MODE_EXCLUSIVE,
    .queueFamilyIndexCount = 0,
    .pQueueFamilyIndices = NULL,
    .presentMode = swapchainPresentMode,
    .oldSwapchain = oldSwapchain,
    .clipped = true,
  };
  err = CreateSwapchainKHR(demo->device, &swapchain, NULL, &demo-
>swapchain);

  // SwapchainのImageを取得します
  VkImage *swapchainImages = (VkImage *)malloc(demo-
>swapchainImageCount * sizeof(VkImage));
  err = demo->fpGetSwapchainImagesKHR(demo->device, demo-
>swapchain,
                                      &demo->swapchainImageCount,
                                      swapchainImages);

  // SwapchainのImageのImageViewを作ります
  for (i = 0; i < demo->swapchainImageCount; i++) {
    VkImageViewCreateInfo color_attachment_view = {
      .sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO,
      .pNext = NULL,
      .format = demo->format,
      .image = swapchainImages[i];
      .components = {
        .r = VK_COMPONENT_SWIZZLE_R,
        .g = VK_COMPONENT_SWIZZLE_G,
        .b = VK_COMPONENT_SWIZZLE_B,
        .a = VK_COMPONENT_SWIZZLE_A },
      .subresourceRange = {.aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT,
                           .baseMipLevel = 0,
                           .levelCount = 1,
                           .baseArrayLayer = 0,
                           .layerCount = 1},
      .viewType = VK_IMAGE_VIEW_TYPE_2D,
      .flags = 0,
    };
    err = vkCreateImageView(demo->device, &color_attachment_view,
NULL, &demo->buffers[i].view);
  }
```

パイプライン

OpenGL のstateとshaderの情報はVulkanでは全部 VkPipeline のオブジェクトに含まれる
Shaderで使うResource (Texture, Uniform Bufferとか)はDescriptionSetで設定する必要がある

```
    // Pipeline生成用の構造体
    VkGraphicsPipelineCreateInfo pipeline;
    VkPipelineInputAssemblyStateCreateInfo ia;
    VkPipelineRasterizationStateCreateInfo rs;
    VkPipelineColorBlendStateCreateInfo cb;
    VkPipelineDepthStencilStateCreateInfo ds;
    VkPipelineViewportStateCreateInfo vp;
    VkDynamicState
dynamicStateEnables[VK_DYNAMIC_STATE_RANGE_SIZE];
    VkPipelineDynamicStateCreateInfo dynamicState;

    // Primitive typeを Triangle List に設定
    ia.sType =
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREAT
E_INFO;
    ia.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

    // Rasterization設定 (Polygon Mode, Culling, Depth Range/Bias設定）
    rs.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_I
NFO;
    rs.polygonMode = VK_POLYGON_MODE_FILL;
    rs.cullMode = VK_CULL_MODE_BACK_BIT;
    rs.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
    rs.depthClampEnable = VK_FALSE;
    rs.rasterizerDiscardEnable = VK_FALSE;
    rs.depthBiasEnable = VK_FALSE;
    rs.lineWidth = 1.0f;

    // Blend、Color Mask設定
    cb.sType =
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_I
NFO;
```

```
    VkPipelin
    memset(att_state, 0, sizeof(att_state));
    att_state[0].colorWriteMask = 0xf;
    att_state[0].blendEnable = VK_FALSE;
    cb.attachmentCount = 1;
    cb.pAttachments = att_state;

    // Viewport と Scissor 設定、ここは動的に変える様に設定します
    vp.sType =
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
    vp.viewportCount = 1;
    dynamicStateEnables[dynamicState.dynamicStateCount++] =
        VK_DYNAMIC_STATE_VIEWPORT;
    vp.scissorCount = 1;
    dynamicStateEnables[dynamicState.dynamicStateCount++] =
        VK_DYNAMIC_STATE_SCISSOR;

    // Depth, Stencil Test設定
    ds.sType =
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_
INFO;
    ds.depthTestEnable = VK_TRUE;
    ds.depthWriteEnable = VK_TRUE;
    ds.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
    ds.depthBoundsTestEnable = VK_FALSE;
    ds.back.failOp = VK_STENCIL_OP_KEEP;
    ds.back.passOp = VK_STENCIL_OP_KEEP;
    ds.back.compareOp = VK_COMPARE_OP_ALWAYS;
    ds.stencilTestEnable = VK_FALSE;
    ds.front = ds.back;

    // Shader stage設定: VSとFS二つのstageを設定
    pipeline.stageCount = 2;
```

```
    memset(&shaderStages, 0, 2 * sizeof(VkPipelineShaderStageCreateInfo));

    // VS設定
    shaderStages[0].sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    shaderStages[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
    shaderStages[0].module = demo_prepare_vs(demo);
    shaderStages[0].pName = "main";

    // FS設定
    shaderStages[1].sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    shaderStages[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    shaderStages[1].module = demo_prepare_fs(demo);
    shaderStages[1].pName = "main";

    // Pipeline生成します
    pipeline.pVertexInputState = &vi;
    pipeline.pInputAssemblyState = &ia;
    pipeline.pRasterizationState = &rs;
    pipeline.pColorBlendState = &cb;
    pipeline.pMultisampleState = &ms;
    pipeline.pViewportState = &vp;
    pipeline.pDepthStencilState = &ds;
    pipeline.pStages = shaderStages;
    pipeline.renderPass = demo->render_pass;
    pipeline.pDynamicState = &dynamicState;
    err = vkCreateGraphicsPipelines(demo->device, demo->pipelineCache, 1,
                                    &pipeline, NULL, &demo->pipeline);
```

DMP DIGITAL MEDIA PROFESSIONALS

Vertex array

```
// vertex data
   const float vb[3][5] = {
      /*     position             texcoord */
      { -1.0f, -1.0f,  0.25f,     0.0f, 0.0f },
      {  1.0f, -1.0f,  0.25f,     1.0f, 0.0f },
      {  0.0f,  1.0f,  1.0f,      0.5f, 1.0f },
   };

   // Bufferを生成します
   err = vkCreateBuffer(demo->device, &buf_info, NULL,
&demo->vertices.buf);

   // メモリ配置
   VkMemoryAllocateInfo mem_alloc;
   mem_alloc.allocationSize = mem_reqs.size;
   vkGetBufferMemoryRequirements(demo->device, demo-
>vertices.buf, &mem_reqs);
   memory_type_from_properties(demo,
mem_reqs.memoryTypeBits,
      VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
      &mem_alloc.memoryTypeIndex);
   err = vkAllocateMemory(demo->device, &mem_alloc,
NULL, &demo->vertices.mem);

   // Vertex Data をコピーします
   err = vkMapMemory(demo->device, demo-
>vertices.mem, 0,
                 mem_alloc.allocationSize, 0, &data);
   memcpy(data, vb, sizeof(vb));
   vkUnmapMemory(demo->device, demo->vertices.mem);

   // Bufferにメモリをアタッチします
   err = vkBindBufferMemory(demo->device, demo-
>vertices.buf,
                 demo->vertices.mem, 0);

   // Description設定
   VkPipelineVertexInputStateCreateInfo vi;
   VkVertexInputBindingDescription vi_bindings[1];
   VkVertexInputAttributeDescription vi_attrs[2];

   // Vertex Buffer  Binding設定
   vi_bindings[0].binding = VERTEX_BUFFER_BIND_ID;
   vi_bindings[0].stride = sizeof(vb[0]);
   vi_bindings[0].inputRate =
VK_VERTEX_INPUT_RATE_VERTEX;

   // Vertex attribute 0設定
   vi_attrs[0].binding = VERTEX_BUFFER_BIND_ID;
   vi_attrs[0].location = 0;
   vi_attrs[0].format =
VK_FORMAT_R32G32B32_SFLOAT;
   vi_attrs[0].offset = 0;

   // Vertex atribute 1設定
   vi_attrs[1].binding = VERTEX_BUFFER_BIND_ID;
   vi_attrs[1].location = 1;
   vi_attrs[1].format = VK_FORMAT_R32G32_SFLOAT;
   vi_attrs[1].offset = sizeof(float) * 3;

   // VkPipelineVertexInputStateCreateInfo設定 vi ->
pipeline生成時使用
   vi.sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_ST
ATE_CREATE_INFO;
   vi.pNext = NULL;
   vi.vertexBindingDescriptionCount = 1;
   vi.pVertexBindingDescriptions = vi_bindings;
   vi.vertexAttributeDescriptionCount = 2;
   vi.pVertexAttributeDescriptions = vi_attrs;
```

Framebuffer

```
// Framebuffer attachment, 0はColor buffer, 1はDepth buffer
VkImageView attachments[2];
attachments[1] = demo->depth.view;
const VkFramebufferCreateInfo fb_info = {
    .sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO,
    .pNext = NULL,
    .renderPass = demo->render_pass,
    .attachmentCount = 2,
    .pAttachments = attachments,
    .width = demo->width,
    .height = demo->height,
    .layers = 1,
};

demo->framebuffers = (VkFramebuffer *)malloc(demo->swapchainImageCount *
                                             sizeof(VkFramebuffer));
// Swapchain Imageの数に合わせてFramebufferを生成します
for (i = 0; i < demo->swapchainImageCount; i++) {
    attachments[0] = demo->buffers[i].view;
    err = vkCreateFramebuffer(demo->device, &fb_info, NULL,
                              &demo->framebuffers[i]);
}
```

レンダーパス

```
// Color attachment
const VkAttachmentReference color_reference = {
    .attachment = 0, .layout =
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
};
// Depth/Stencil attachment
const VkAttachmentReference depth_reference = {
    .attachment = 1,
    .layout =
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
};
// Subpass情報
const VkSubpassDescription subpass = {
    .pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS,
    .flags = 0,
    .inputAttachmentCount = 0,
    .pInputAttachments = NULL,
    .colorAttachmentCount = 1,
    .pColorAttachments = &color_reference,
    .pResolveAttachments = NULL,
    .pDepthStencilAttachment = &depth_reference,
    .preserveAttachmentCount = 0,
    .pPreserveAttachments = NULL,
};
// Render Pass attachment情報
const VkAttachmentDescription attachments[2] = {
    [0] =
      {
       .format = demo->format,
       .samples = VK_SAMPLE_COUNT_1_BIT,
       .loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR,
       .storeOp = VK_ATTACHMENT_STORE_OP_STORE,
       .stencilLoadOp =
VK_ATTACHMENT_LOAD_OP_DONT_CARE,
       .stencilStoreOp =
VK_ATTACHMENT_STORE_OP_DONT_CARE,
       .initialLayout =
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
       .finalLayout =
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
      },
    [1] =
      {
       .format = demo->depth.format,
       .samples = VK_SAMPLE_COUNT_1_BIT,
       .loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR,
       .storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE,
       .stencilLoadOp =
VK_ATTACHMENT_LOAD_OP_DONT_CARE,
       .stencilStoreOp =
VK_ATTACHMENT_STORE_OP_DONT_CARE,
       .initialLayout =

VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
       .finalLayout =

VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL,
      },
};
// Render Pass情報
const VkRenderPassCreateInfo rp_info = {
    .sType =
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO,
    .pNext = NULL,
    .attachmentCount = 2,
    .pAttachments = attachments,
    .subpassCount = 1,
    .pSubpasses = &subpass,
    .dependencyCount = 0,
    .pDependencies = NULL,
};

// Render Passを生成します
err = vkCreateRenderPass(demo->device, &rp_info, NULL, &demo-
>render_pass);
```

Texture

- OpenGLではtexture設定する時、ドライバは自動的にHWのtilingとlayoutに変換するが、Vulkanでは全部手動でやる必要
- Texture設定の手順：
    1. 設定用のimage、tiling=VK_IMAGE_TILING_LINEARを設定して作成 (VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT の属性を持つメモリを使う必要)
    2. 設定用のimage, layoutをVK_IMAGE_LAYOUT_PREINITIALIZEDに設定して、texture dataをコピー
    3. 描画用のimage、VK_IMAGE_TILING_OPTIMALを設定して作成
    4. 設定用のimageのlayoutをVK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL に変更。また、描画用のimageのlayoutをVK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMALに変更
    5. vkCmdCopyImageでdataを設定用のimageから描画用のimageにコピー
    6. 描画用のimageのlayoutを実際使用のlayoutに変更（e.g. VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL）

Texture – Image生成

```
// Image生成情報
const VkImageCreateInfo image_create_info = {
    .sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,
    .pNext = NULL,
    .imageType = VK_IMAGE_TYPE_2D,
    .format = tex_format,
    .extent = {tex_width, tex_height, 1},
    .mipLevels = 1,
    .arrayLayers = 1,
    .samples = VK_SAMPLE_COUNT_1_BIT,
    .tiling = tiling,
    .usage = usage,
    .flags = 0,
    .initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED
};
VkMemoryAllocateInfo mem_alloc = {
    .sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO,
    .pNext = NULL,
    .allocationSize = 0,
    .memoryTypeIndex = 0,
};
VkMemoryRequirements mem_reqs;
```

```
// Imageを生成します
err = vkCreateImage(demo->device, &image_create_info, NULL, &tex_obj->image);

// Imageを使用するMemory属性を探します
vkGetImageMemoryRequirements(demo->device, tex_obj->image, &mem_reqs);
mem_alloc.allocationSize = mem_reqs.size;
pass = memory_type_from_properties(demo, mem_reqs.memoryTypeBits,
                                   required_props, &mem_alloc.memoryTypeIndex);

// Memory配置
err = vkAllocateMemory(demo->device, &mem_alloc, NULL, &tex_obj->mem);

// ImageにMemoryをattach
err = vkBindImageMemory(demo->device, tex_obj->image, tex_obj->mem, 0);
```

Texture – Image Data設定

```
// Image Data設定
const VkImageSubresource subres = {
     .aspectMask = VK_IMAGE_ASPECT_COLOR_BIT,
     .mipLevel = 0,
     .arrayLayer = 0,
 };
 VkSubresourceLayout layout;
 void *data;
 int32_t x, y;

 // SubImageのlayout情報を取得
 vkGetImageSubresourceLayout(demo->device, tex_obj->image, &subres, &layout);

 // Map Memory
 err = vkMapMemory(demo->device, tex_obj->mem, 0, mem_alloc.allocationSize, 0, &data);

 // Texture Dataをコピーします
 memcpy(data, texure_data, data_size);

 // Unmap   Memory
 vkUnmapMemory(demo->device, tex_obj->mem);
```

Texture – Image Layout変更

```
// Image layoutのはimage meory barrierの操作で変更します
   VkImageMemoryBarrier image_memory_barrier = {
      .sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,
      .pNext = NULL,
      .srcAccessMask = srcAccessMask,
      .dstAccessMask = dstAccessMask,
      .oldLayout = old_image_layout,  // 変更前のlayout
      .newLayout = new_image_layout,  // 変更する layout
      .image = image,
      .subresourceRange = {aspectMask, 0, 1, 0, 1}};

   VkPipelineStageFlags src_stages = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
   VkPipelineStageFlags dest_stages = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;

   // Barrierをコマンドバファに入れます
   vkCmdPipelineBarrier(demo->setup_cmd, src_stages, dest_stages, 0, 0, NULL, 0, NULL, 1, &image_memory_barrier);
vkUnmapMemory(demo->device, tex_obj->mem);
```

Texture – Image Dataコピー

```
VkImageCopy copy_region = {
      .srcSubresource = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 0, 1},
      .srcOffset = {0, 0, 0},
      .dstSubresource = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 0, 1},
      .dstOffset = {0, 0, 0},
      .extent = {staging_texture.tex_width,
                 staging_texture.tex_height, 1},
  };
  vkCmdCopyImage(
      demo->setup_cmd, staging_texture.image,
      VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, demo->textures[i].image,
      VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &copy_region);

  // ここでは省略しましたが、
  // このコマンドもコマンドキューに入れないと実際にコピーしません
```

Texture – Sampler

```
const VkSamplerCreateInfo sampler = {
    .sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO,
    .pNext = NULL,
    .magFilter = VK_FILTER_NEAREST,
    .minFilter = VK_FILTER_NEAREST,
    .mipmapMode = VK_SAMPLER_MIPMAP_MODE_NEAREST,
    .addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT,
    .addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT,
    .addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT,
    .mipLodBias = 0.0f,
    .anisotropyEnable = VK_FALSE,
    .maxAnisotropy = 1,
    .compareOp = VK_COMPARE_OP_NEVER,
    .minLod = 0.0f,
    .maxLod = 0.0f,
    .borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE,
    .unnormalizedCoordinates = VK_FALSE,
};
err = vkCreateSampler(demo->device, &sampler, NULL,
                      &demo->textures[i].sampler);
```

Texture – ImageView

```
VkImageViewCreateInfo view = {
        .sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO,
        .pNext = NULL,
        .image = demo->textures[i].image,
        .viewType = VK_IMAGE_VIEW_TYPE_2D,
        .format = tex_format,
        .components =
            {
             VK_COMPONENT_SWIZZLE_R, VK_COMPONENT_SWIZZLE_G,
             VK_COMPONENT_SWIZZLE_B, VK_COMPONENT_SWIZZLE_A,
            },
        .subresourceRange = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1},
        .flags = 0,
    };
    err = vkCreateImageView(demo->device, &view, NULL,
                            &demo->textures[i].view);
```

描画

- Vulkanにおける描画
  - まずコマンドバファ作成
  - このコマンドバファをコマンドキューにenqueuすることで描画

描画 – コマンドバッファを作る

```
    const VkCommandBufferInheritanceInfo cmd_buf_hinfo = {
        .sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO,
        .pNext = NULL,
        .renderPass = VK_NULL_HANDLE,
        .subpass = 0,
        .framebuffer = VK_NULL_HANDLE,
        .occlusionQueryEnable = VK_FALSE,
        .queryFlags = 0,
        .pipelineStatistics = 0,
    };
    const VkCommandBufferBeginInfo cmd_buf_info = {
        .sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO,
        .pNext = NULL,
        .flags = 0,
        .pInheritanceInfo = &cmd_buf_hinfo,
    };
    const VkClearValue clear_values[2] = {
            [0] = {.color.float32 = {0.2f, 0.2f, 0.2f, 0.2f}},
            [1] = {.depthStencil = {demo->depthStencil, 0}},
    };
     // この描画のRender Passの情報
    const VkRenderPassBeginInfo rp_begin = {
        .sType =
VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO,
        .pNext = NULL,
        .renderPass = demo->render_pass,
        .framebuffer = demo->framebuffers[demo->current_buffer],
//現在の描画先のFramebuffer
        .renderArea.offset.x = 0,
        .renderArea.offset.y = 0,
        .renderArea.extent.width = demo->width,
        .renderArea.extent.height = demo->height,
        .clearValueCount = 2,
        .pClearValues = clear_values,
    };

    // コマンドバファ生成開始
    err = vkBeginCommandBuffer(demo->draw_cmd, &cmd_buf_info);

    // We can use LAYOUT_UNDEFINED as a wildcard here because
we don't care what
    // happens to the previous contents of the image
    VkImageMemoryBarrier image_memory_barrier = {
        .sType =
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,
        .pNext = NULL,
        .srcAccessMask = 0,
        .dstAccessMask =
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
        .oldLayout = VK_IMAGE_LAYOUT_UNDEFINED,
        .newLayout =
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
        .srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
        .dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
        .image = demo->buffers[demo->current_buffer].image,
        .subresourceRange = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 1,
0, 1}};
    //描画先のImageLayoutを
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMALに変更
    vkCmdPipelineBarrier(demo->draw_cmd,
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
                         VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, 0,
0, NULL, 0,
                         NULL, 1, &image_memory_barrier);

    // RenderPass開始を宣告
    // glBindFramebuffer相当
    vkCmdBeginRenderPass(demo->draw_cmd, &rp_begin,
VK_SUBPASS_CONTENTS_INLINE);

    // 描画用のPipelineをbindします
    // glUseProgram, そして描画のstateに相当
    vkCmdBindPipeline(demo->draw_cmd,
VK_PIPELINE_BIND_POINT_GRAPHICS,   demo->pipeline);

    // 描画用のResource情報のDescriptorSetをbindします
    // glBindVertexArray, glBindTexture*, glBindBufferBaseとかに
相当
    vkCmdBindDescriptorSets(demo->draw_cmd,
VK_PIPELINE_BIND_POINT_GRAPHICS,
                          demo->pipeline_layout, 0, 1, &demo->desc_set, 0,
                          NULL);
```

描画 – コマンドバッファを作る(cont'd)

```
// Viewportを設定します
VkViewport viewport;
memset(&viewport, 0, sizeof(viewport));
viewport.height = (float)demo->height;
viewport.width = (float)demo->width;
viewport.minDepth = (float)0.0f;
viewport.maxDepth = (float)1.0f;
vkCmdSetViewport(demo->draw_cmd, 0, 1, &viewport);

// Scissorを設定します
VkRect2D scissor;
memset(&scissor, 0, sizeof(scissor));
scissor.extent.width = demo->width;
scissor.extent.height = demo->height;
scissor.offset.x = 0;
scissor.offset.y = 0;
vkCmdSetScissor(demo->draw_cmd, 0, 1, &scissor);

// 描画のVertex DataのBufferをbindします
// glBindBuffer(GL_VERTEX_ARRAY,...)に相当
VkDeviceSize offsets[1] = {0};
vkCmdBindVertexBuffers(demo->draw_cmd,
VERTEX_BUFFER_BIND_ID, 1,
            &demo->vertices.buf, offsets);

// 描画コマンドを発行します
// glDrawArraysに相当
vkCmdDraw(demo->draw_cmd, 3, 1, 0, 0);

// RenderPass終了
vkCmdEndRenderPass(demo->draw_cmd);

// 描画ようのImageのImageLayoutを
VK_IMAGE_LAYOUT_PRESENT_SRC_KHRに変更します
// (表示のためのBarrier)
VkImageMemoryBarrier prePresentBarrier = {
    .sType =
VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER,
    .pNext = NULL,
    .srcAccessMask =
VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT,
    .dstAccessMask =
VK_ACCESS_MEMORY_READ_BIT,
    .oldLayout =
VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL,
    .newLayout =
VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,
    .srcQueueFamilyIndex =
VK_QUEUE_FAMILY_IGNORED,
    .dstQueueFamilyIndex =
VK_QUEUE_FAMILY_IGNORED,
    .subresourceRange =
{VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1}
    .image = demo->buffers[demo->current_buffer].image;
  };
  vkCmdPipelineBarrier(demo->draw_cmd,
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,

VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, 0, 0,
NULL, 0,
            NULL, 1, &prePresentBarrier);

  // コマンドバファ生成終了
  err = vkEndCommandBuffer(demo->draw_cmd);
```

コマンドバファをコマンドキューにEnqueして描画

```
// 表示完了待ちのSemaphoreオブジェクトを作ります
VkSemaphore presentCompleteSemaphore;
VkSemaphoreCreateInfo presentCompleteSemaphoreCreateInfo = {
    .sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO,
    .pNext = NULL,
    .flags = 0,
};
err = vkCreateSemaphore(demo->device, &presentCompleteSemaphoreCreateInfo,
                        NULL, &presentCompleteSemaphore);

// 次の描画するSwapchain Imageのindexを取得します
err = AcquireNextImageKHR(demo->device, demo->swapchain, UINT64_MAX,
                          presentCompleteSemaphore,
                          (VkFence)0, // TODO: Show use of fence
                          &demo->current_buffer);

// Semaphoreをまつことで、表示完了を待ちます
VkPipelineStageFlags pipe_stage_flags = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkSubmitInfo submit_info = {.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO,
                            .pNext = NULL,
                            .waitSemaphoreCount = 1,
                            .pWaitSemaphores = &presentCompleteSemaphore,   // このコマンドを実行する前に待つSemaphore
                            .pWaitDstStageMask = &pipe_stage_flags,
                            .commandBufferCount = 1,
                            .pCommandBuffers = &demo->draw_cmd,         // コマンドバファのhandle
                            .signalSemaphoreCount = 0,
                            .pSignalSemaphores = NULL};
// コマンドキューにコマンドバファをEnqueueします
// glFlush相当
err = vkQueueSubmit(demo->queue, 1, &submit_info, VK_NULL_HANDLE);

// 描画の結果を表示します
VkPresentInfoKHR present = {
    .sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR,
    .pNext = NULL,
    .swapchainCount = 1,
    .pSwapchains = &demo->swapchain,
    .pImageIndices = &demo->current_buffer,
};
// eglSwapBuffers相当
err = QueuePresentKHR(demo->queue, &present);

// 実行完了を待ちます
// glFinish相当
err = vkQueueWaitIdle(demo->queue);

// 完了待ち用のSemaphoreを削除うします
vkDestroySemaphore(demo->device, presentCompleteSemaphore, NULL);
```

参考

- https://developer.nvidia.com/engaging-voyage-vulkan
- https://www.khronos.org/developers/library/2016-vulkan-devday-uk
- http://on-demand.gputechconf.com/gtc/2016/events/vulkanday/Migrating_from_OpenGL_to_Vulkan.pdf