CODE GENERATION FOR VULKAN INITIALIZATION.

LEARNING VULKAN

- So…
- I’ve decided to learn Vulkan.
- Did I learn it?
- Not yet.

WHAT DID I LEARN?

THIS:

```
// Copyright(c) 2018-2019, NVIDIA CORPORATION. All rights reserved.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//
// VulkanHpp Samples : 05_InitSwapchainRAII
//                     Initialize a swapchain

#include "../utils/utils.hpp"

#include <iostream>

static char const * AppName    = "05_InitSwapchainRAII";
static char const * EngineName = "Vulkan.hpp";

int main( int /*argc*/, char ** /*argv*/ )
{
  try
  {
    vk::raii::Context  context;
    vk::raii::Instance instance =
      vk::raii::su::makeInstance( context, AppName, EngineName, {}, vk::su::getInstanceExtensions() );
#if !defined( NDEBUG )
    vk::raii::DebugUtilsMessengerEXT debugUtilsMessenger( instance,
vk::su::makeDebugUtilsMessengerCreateInfoEXT() );
#endif
    vk::raii::PhysicalDevice physicalDevice = std::move( vk::raii::PhysicalDevices( instance ).front() );

    std::vector<vk::QueueFamilyProperties> queueFamilyProperties = physicalDevice.getQueueFamilyProperties();
    uint32_t graphicsQueueFamilyIndex = vk::su::findGraphicsQueueFamilyIndex( queueFamilyProperties );

    /* VULKAN_HPP_KEY_START */

    uint32_t          width  = 64;
    uint32_t          height = 64;
    vk::su::WindowData window = vk::su::createWindow( AppName, { width, height } );
    VkSurfaceKHR      _surface;
    glfwCreateWindowSurface( static_cast<VkInstance>( *instance ), window.handle, nullptr, &_surface );
    vk::raii::SurfaceKHR surface( instance, _surface );

    // determine a queueFamilyIndex that suports present
    // first check if the graphicsQueueFamilyIndex is good enough
    uint32_t presentQueueFamilyIndex = physicalDevice.getSurfaceSupportKHR( graphicsQueueFamilyIndex, *surface )
                                       ? graphicsQueueFamilyIndex
                                       : vk::su::checked_cast<uint32_t>( queueFamilyProperties.size() );
    if ( presentQueueFamilyIndex == queueFamilyProperties.size() )
    {
      // the graphicsQueueFamilyIndex doesn't support present -> look for an other family index that supports
both
      // graphics and present
      for ( size_t i = 0; i < queueFamilyProperties.size(); i++ )
      {
        if ( ( queueFamilyProperties[i].queueFlags & vk::QueueFlagBits::eGraphics ) &&
             physicalDevice.getSurfaceSupportKHR( vk::su::checked_cast<uint32_t>( i ), *surface ) )
        {
          graphicsQueueFamilyIndex = vk::su::checked_cast<uint32_t>( i );
          presentQueueFamilyIndex  = graphicsQueueFamilyIndex;
          break;
        }
      }
      if ( presentQueueFamilyIndex == queueFamilyProperties.size() )
      {
        // there's nothing like a single family index that supports both graphics and present -> look for an
other
        // family index that supports present
        for ( size_t i = 0; i < queueFamilyProperties.size(); i++ )
        {
          if ( physicalDevice.getSurfaceSupportKHR( vk::su::checked_cast<uint32_t>( i ), *surface ) )
          {
            presentQueueFamilyIndex = vk::su::checked_cast<uint32_t>( i );
            break;
          }
        }
      }
    }
    if ( ( graphicsQueueFamilyIndex == queueFamilyProperties.size() ) ||
         ( presentQueueFamilyIndex == queueFamilyProperties.size() ) )
    {
      throw std::runtime_error( "Could not find a queue for graphics or present -> terminating" );
    }

    // create a device
    vk::raii::Device device =
      vk::raii::su::makeDevice( physicalDevice, graphicsQueueFamilyIndex, vk::su::getDeviceExtensions() );

    // get the supported VkFormats
    std::vector<vk::SurfaceFormatKHR> formats = physicalDevice.getSurfaceFormatsKHR( *surface );
    assert( !formats.empty() );
    vk::Format format =
      ( formats[0].format == vk::Format::eUndefined ) ? vk::Format::eB8G8R8A8Unorm : formats[0].format;

    vk::SurfaceCapabilitiesKHR surfaceCapabilities = physicalDevice.getSurfaceCapabilitiesKHR( *surface );
    VkExtent2D                 swapchainExtent;
    if ( surfaceCapabilities.currentExtent.width == std::numeric_limits<uint32_t>::max() )
    {
      // If the surface size is undefined, the size is set to the size of the images requested.
      swapchainExtent.width =
        vk::su::clamp( width, surfaceCapabilities.minImageExtent.width, surfaceCapabilities.maxImageExtent.width
);
      swapchainExtent.height =
        vk::su::clamp( height, surfaceCapabilities.minImageExtent.height,
surfaceCapabilities.maxImageExtent.height );
    }
    else
    {
      // If the surface size is defined, the swap chain size must match
      swapchainExtent = surfaceCapabilities.currentExtent;
    }

    // The FIFO present mode is guaranteed by the spec to be supported
    vk::PresentModeKHR swapchainPresentMode = vk::PresentModeKHR::eFifo;

    vk::SurfaceTransformFlagBitsKHR preTransform =
      ( surfaceCapabilities.supportedTransforms & vk::SurfaceTransformFlagBitsKHR::eIdentity )
        ? vk::SurfaceTransformFlagBitsKHR::eIdentity
        : surfaceCapabilities.currentTransform;

    vk::CompositeAlphaFlagBitsKHR compositeAlpha =
      ( surfaceCapabilities.supportedCompositeAlpha & vk::CompositeAlphaFlagBitsKHR::ePreMultiplied )
        ? vk::CompositeAlphaFlagBitsKHR::ePreMultiplied
      : ( surfaceCapabilities.supportedCompositeAlpha & vk::CompositeAlphaFlagBitsKHR::ePostMultiplied )
        ? vk::CompositeAlphaFlagBitsKHR::ePostMultiplied
      : ( surfaceCapabilities.supportedCompositeAlpha & vk::CompositeAlphaFlagBitsKHR::eInherit )
        ? vk::CompositeAlphaFlagBitsKHR::eInherit
        : vk::CompositeAlphaFlagBitsKHR::eOpaque;

    vk::SwapchainCreateInfoKHR swapChainCreateInfo( vk::SwapchainCreateFlagsKHR(),
                                                    *surface,
                                                    surfaceCapabilities.minImageCount,
                                                    format,
                                                    vk::ColorSpaceKHR::eSrgbNonlinear,
                                                    swapchainExtent,
                                                    1,
                                                    vk::ImageUsageFlagBits::eColorAttachment,
                                                    vk::SharingMode::eExclusive,
                                                    {},
                                                    preTransform,
                                                    compositeAlpha,
                                                    swapchainPresentMode,
                                                    true,
                                                    nullptr );

    std::array<uint32_t, 2> queueFamilyIndices = { graphicsQueueFamilyIndex, presentQueueFamilyIndex };
    if ( graphicsQueueFamilyIndex != presentQueueFamilyIndex )
    {
      // If the graphics and present queues are from different queue families, we either have to explicitly
transfer
      // ownership of images betwees the queues, or we have to create the swapchain with imageSharingMode as
      // VK_SHARING_MODE_CONCURRENT
      swapChainCreateInfo.imageSharingMode      = vk::SharingMode::eConcurrent;
      swapChainCreateInfo.queueFamilyIndexCount = vk::su::checked_cast<uint32_t>( queueFamilyIndices.size() );
      swapChainCreateInfo.pQueueFamilyIndices   = queueFamilyIndices.data();
    }

    vk::raii::SwapchainKHR swapChain( device, swapChainCreateInfo );
    std::vector<VkImage>   swapChainImages = swapChain.getImages();

    std::vector<vk::raii::ImageView> imageViews;
    imageViews.reserve( swapChainImages.size() );
    vk::ComponentMapping componentMapping(
      vk::ComponentSwizzle::eR, vk::ComponentSwizzle::eG, vk::ComponentSwizzle::eB, vk::ComponentSwizzle::eA );
    vk::ImageSubresourceRange subResourceRange( vk::ImageAspectFlagBits::eColor, 0, 1, 0, 1 );
    for ( auto image : swapChainImages )
    {
      vk::ImageViewCreateInfo imageViewCreateInfo(
        {}, static_cast<vk::Image>( image ), vk::ImageViewType::e2D, format, componentMapping, subResourceRange
);
      imageViews.push_back( { device, imageViewCreateInfo } );
    }

    /* VULKAN_HPP_KEY_END */
  }
  catch ( vk::SystemError & err )
  {
    std::cout << "vk::SystemError: " << err.what() << std::endl;
    exit( -1 );
  }
  catch ( std::exception & err )
  {
  }
```

THIS IS JUST INITIALIZATION

```
// Copyright(c) 2018-2019, NVIDIA CORPORATION. All rights reserved.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//
// VulkanHpp Samples : 05_InitSwapchainRAII
//                     Initialize a swapchain

#include "../utils/utils.hpp"

#include <iostream>

static char const * AppName    = "05_InitSwapchainRAII";
static char const * EngineName = "Vulkan.hpp";

int main( int /*argc*/, char ** /*argv*/ )
{
  try
  {
    vk::raii::Context  context;
    vk::raii::Instance instance =
      vk::raii::su::makeInstance( context, AppName, EngineName, {}, vk::su::getInstanceExtensions() );
#if !defined( NDEBUG )
    vk::raii::DebugUtilsMessengerEXT debugUtilsMessenger( instance,
vk::su::makeDebugUtilsMessengerCreateInfoEXT() );
#endif
    vk::raii::PhysicalDevice physicalDevice = std::move( vk::raii::PhysicalDevices( instance ).front() );

    std::vector<vk::QueueFamilyProperties> queueFamilyProperties = physicalDevice.getQueueFamilyProperties();
    uint32_t graphicsQueueFamilyIndex = vk::su::findGraphicsQueueFamilyIndex( queueFamilyProperties );

    /* VULKAN_HPP_KEY_START */

    uint32_t          width  = 64;
    uint32_t          height = 64;
    vk::su::WindowData window = vk::su::createWindow( AppName, { width, height } );
    VkSurfaceKHR      _surface;
    glfwCreateWindowSurface( static_cast<VkInstance>( *instance ), window.handle, nullptr, &_surface );
    vk::raii::SurfaceKHR surface( instance, _surface );

    // determine a queueFamilyIndex that suports present
    // first check if the graphicsQueueFamilyIndex is good enough
    uint32_t presentQueueFamilyIndex = physicalDevice.getSurfaceSupportKHR( graphicsQueueFamilyIndex, *surface )
                                       ? graphicsQueueFamilyIndex
                                       : vk::su::checked_cast<uint32_t>( queueFamilyProperties.size() );
    if ( presentQueueFamilyIndex == queueFamilyProperties.size() )
    {
      // the graphicsQueueFamilyIndex doesn't support present -> look for an other family index that supports
both
      // graphics and present
      for ( size_t i = 0; i < queueFamilyProperties.size(); i++ )
      {
        if ( ( queueFamilyProperties[i].queueFlags & vk::QueueFlagBits::eGraphics ) &&
             physicalDevice.getSurfaceSupportKHR( vk::su::checked_cast<uint32_t>( i ), *surface ) )
        {
          graphicsQueueFamilyIndex = vk::su::checked_cast<uint32_t>( i );
          presentQueueFamilyIndex  = graphicsQueueFamilyIndex;
          break;
        }
      }
      if ( presentQueueFamilyIndex == queueFamilyProperties.size() )
      {
        // there's nothing like a single family index that supports both graphics and present -> look for an
other
        // family index that supports present
        for ( size_t i = 0; i < queueFamilyProperties.size(); i++ )
        {
          if ( physicalDevice.getSurfaceSupportKHR( vk::su::checked_cast<uint32_t>( i ), *surface ) )
          {
            presentQueueFamilyIndex = vk::su::checked_cast<uint32_t>( i );
            break;
          }
        }
      }
    }
    if ( ( graphicsQueueFamilyIndex == queueFamilyProperties.size() ) ||
         ( presentQueueFamilyIndex == queueFamilyProperties.size() ) )
    {
      throw std::runtime_error( "Could not find a queue for graphics or present -> terminating" );
    }

    // create a device
    vk::raii::Device device =
      vk::raii::su::makeDevice( physicalDevice, graphicsQueueFamilyIndex, vk::su::getDeviceExtensions() );

    // get the supported VkFormats
    std::vector<vk::SurfaceFormatKHR> formats = physicalDevice.getSurfaceFormatsKHR( *surface );
    assert( !formats.empty() );
    vk::Format format =
      ( formats[0].format == vk::Format::eUndefined ) ? vk::Format::eB8G8R8A8Unorm : formats[0].format;

    vk::SurfaceCapabilitiesKHR surfaceCapabilities = physicalDevice.getSurfaceCapabilitiesKHR( *surface );
    VkExtent2D                 swapchainExtent;
    if ( surfaceCapabilities.currentExtent.width == std::numeric_limits<uint32_t>::max() )
    {
      // If the surface size is undefined, the size is set to the size of the images requested.
      swapchainExtent.width =
        vk::su::clamp( width, surfaceCapabilities.minImageExtent.width, surfaceCapabilities.maxImageExtent.width
);
      swapchainExtent.height =
        vk::su::clamp( height, surfaceCapabilities.minImageExtent.height,
surfaceCapabilities.maxImageExtent.height );
    }
    else
    {
      // If the surface size is defined, the swap chain size must match
      swapchainExtent = surfaceCapabilities.currentExtent;
    }

    // The FIFO present mode is guaranteed by the spec to be supported
    vk::PresentModeKHR swapchainPresentMode = vk::PresentModeKHR::eFifo;

    vk::SurfaceTransformFlagBitsKHR preTransform =
      ( surfaceCapabilities.supportedTransforms & vk::SurfaceTransformFlagBitsKHR::eIdentity )
        ? vk::SurfaceTransformFlagBitsKHR::eIdentity
        : surfaceCapabilities.currentTransform;

    vk::CompositeAlphaFlagBitsKHR compositeAlpha =
      ( surfaceCapabilities.supportedCompositeAlpha & vk::CompositeAlphaFlagBitsKHR::ePreMultiplied )
        ? vk::CompositeAlphaFlagBitsKHR::ePreMultiplied
      : ( surfaceCapabilities.supportedCompositeAlpha & vk::CompositeAlphaFlagBitsKHR::ePostMultiplied )
        ? vk::CompositeAlphaFlagBitsKHR::ePostMultiplied
      : ( surfaceCapabilities.supportedCompositeAlpha & vk::CompositeAlphaFlagBitsKHR::eInherit )
        ? vk::CompositeAlphaFlagBitsKHR::eInherit
        : vk::CompositeAlphaFlagBitsKHR::eOpaque;

    vk::SwapchainCreateInfoKHR swapChainCreateInfo( vk::SwapchainCreateFlagsKHR(),
                                                    *surface,
                                                    surfaceCapabilities.minImageCount,
                                                    format,
                                                    vk::ColorSpaceKHR::eSrgbNonlinear,
                                                    swapchainExtent,
                                                    1,
                                                    vk::ImageUsageFlagBits::eColorAttachment,
                                                    vk::SharingMode::eExclusive,
                                                    {},
                                                    preTransform,
                                                    compositeAlpha,
                                                    swapchainPresentMode,
                                                    true,
                                                    nullptr );

    std::array<uint32_t, 2> queueFamilyIndices = { graphicsQueueFamilyIndex, presentQueueFamilyIndex };
    if ( graphicsQueueFamilyIndex != presentQueueFamilyIndex )
    {
      // If the graphics and present queues are from different queue families, we either have to explicitly
transfer
      // ownership of images between the queues, or we have to create the swapchain with imageSharingMode as
      // VK_SHARING_MODE_CONCURRENT
      swapChainCreateInfo.imageSharingMode      = vk::SharingMode::eConcurrent;
      swapChainCreateInfo.queueFamilyIndexCount = vk::su::checked_cast<uint32_t>( queueFamilyIndices.size() );
      swapChainCreateInfo.pQueueFamilyIndices   = queueFamilyIndices.data();
    }

    vk::raii::SwapchainKHR swapChain( device, swapChainCreateInfo );
    std::vector<VkImage>   swapChainImages = swapChain.getImages();

    std::vector<vk::raii::ImageView> imageViews;
    imageViews.reserve( swapChainImages.size() );
    vk::ComponentMapping componentMapping(
      vk::ComponentSwizzle::eR, vk::ComponentSwizzle::eG, vk::ComponentSwizzle::eB, vk::ComponentSwizzle::eA );
    vk::ImageSubresourceRange subResourceRange( vk::ImageAspectFlagBits::eColor, 0, 1, 0, 1 );
    for ( auto image : swapChainImages )
    {
      vk::ImageViewCreateInfo imageViewCreateInfo(
        {}, static_cast<vk::Image>( image ), vk::ImageViewType::e2D, format, componentMapping, subResourceRange
);
      imageViews.push_back( { device, imageViewCreateInfo } );
    }

    /* VULKAN_HPP_KEY_END */
  }
  catch ( vk::SystemError & err )
  {
    std::cout << "vk::SystemError: " << err.what() << std::endl;
    exit( -1 );
  }
  catch ( std::exception & err )
  {
  }
```

INITIALIZING WHAT?

1. Initialize Instance
2. Select physical device
3. Create logical device + queues
4. Create a presentation queue
5. Create a Swap chain

Don’t forget to select the correct extensions.

VULKAN API REGISTRY

- XML file
- automatic header generation

```
<?xml version="1.0" encoding="UTF-8"?>
<registry>
    <comment>
Copyright 2015-2021 The Khronos Group Inc.
SPDX-License-Identifier: Apache-2.0 OR MIT
    </comment>
…
    <commands comment="Vulkan command definitions">
        <command successcodes="VK_SUCCESS"
errorcodes="VK_ERROR_OUT_OF_HOST_MEMORY,VK_ERROR_OUT_OF_DEVICE_MEMORY,VK_ERROR_INITIALIZATION_FAILED,VK_ERROR_LAY
ER_NOT_PRESENT,VK_ERROR_EXTENSION_NOT_PRESENT,VK_ERROR_INCOMPATIBLE_DRIVER">
            <proto><type>VkResult</type> <name>vkCreateInstance</name></proto>
            <param>const <type>VkInstanceCreateInfo</type>* <name>pCreateInfo</name></param>
            <param optional="true">const <type>VkAllocationCallbacks</type>* <name>pAllocator</name></param>
            <param><type>VkInstance</type>* <name>pInstance</name></param>
        </command>
…
    </commands>
…
</registry>
```

VULKAN API REGISTRY

This generates this :

```
template <typename Dispatch>
VULKAN_HPP_NODISCARD VULKAN_HPP_INLINE Result
createInstance(
   const VULKAN_HPP_NAMESPACE::InstanceCreateInfo *   pCreateInfo,
   const VULKAN_HPP_NAMESPACE::AllocationCallbacks *  pAllocator,
   VULKAN_HPP_NAMESPACE::Instance *                   pInstance,
   Dispatch const &                                   d
) VULKAN_HPP_NOEXCEPT
```

WHAT ELSE IS THERE?

- All types
- All platforms
- `#define` platform selection
- All extensions

DEPENDENCIES

WHAT DID I DO?

- Simple code generation
  - XSLT translation to header file
- Enumerate all the things
  - Platforms, extensions, and types
- `constexpr` Type traits
  - Association between elements

WHAT ARE MY PLANS?

- Simplify initialization using `Builder` classes
  - The `Builder` class knows it's pre-requisites
- `Builder` for target object create required predecessors
  - Simplifying the generation of requirements
- Automate extension and device selection
  - `Builder` object, selects extensions based on targeted objective.

THANK YOU!

You can find this at:

https://github.com/bogado/Vulkan_tests

Victor Bogado

https://www.bogado.net/

Twitter: @bogado

email: victor@bogado.net