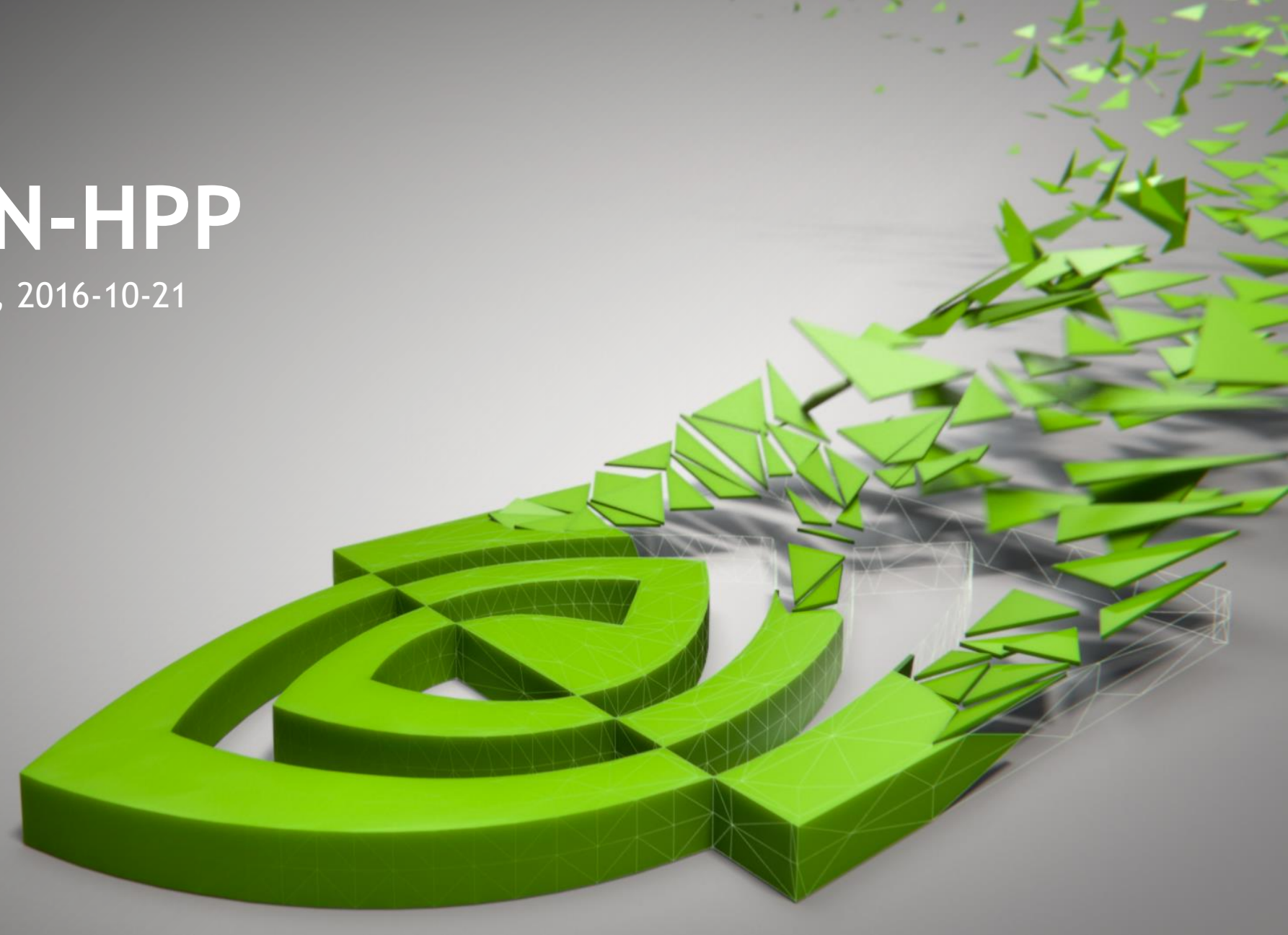
VULKAN-HPP

Markus Tavenrath, 2016-10-21

NVIDIA



WHAT IS VULKAN-HPP

Vulkan-Hpp is an autogenerated C++11 binding for Vulkan with the following goals

1. Improve error detection at compile time
2. Shorten source code which interacts with Vulkan API
3. Simplify error handling

VULKAN NAMESPACE

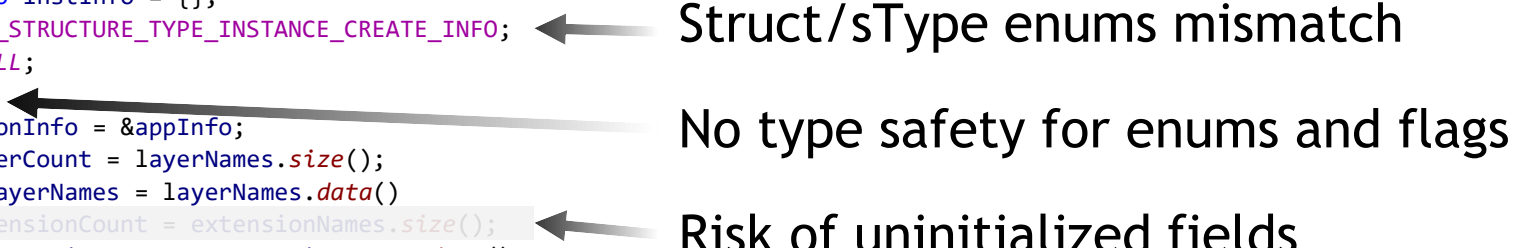
```
// Strip Vk prefix of all functions and structs
VkResult vkCreateInstance(const VkInstanceCreateInfo* pCreateInfo,
                          const VkAllocationCallbacks* pAllocator,
                          VkInstance* pInstance);
```


```
// Introduce new vk namespace for all Vulkan-Hpp symbols
namespace vk {
  Result createInstance(const InstanceCreateInfo* pCreateInfo,
                        const AllocationCallbacks* pAllocator,
                        Instance* pInstance);
};
```

COMPILE TIME ERROR DETECTION

```
VkInstanceCreateInfo instInfo = {};
instInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
instInfo.pNext = NULL;
instInfo.flags = 0;
instInfo.pApplicationInfo = &appInfo;
instInfo.enabledLayerCount = layerNames.size();
instInfo.ppEnabledLayerNames = layerNames.data()
instInfo.enabledExtensionCount = extensionNames.size();
instInfo.ppEnabledExtensionNames = extensionNames.data();

VkResult res = vkCreateInstance(&instInfo, NULL, &info.inst);
assert(res == VK_SUCCESS);
```



TYPE SAFETY

Enums

```
namespace vk
 {
  // Use scoped enums for type safety
  enum class ImageType
  {
    e1D = VK_IMAGE_TYPE_1D,
    e2D = VK_IMAGE_TYPE_2D,
    e3D = VK_IMAGE_TYPE_3D
  };
}
```

‘e‘ + name

Strip VK_ prefix + enum name

Use upper camel case of enum type as name

prefix is required only for numbers, used everywhere for consistency reasons

TYPE SAFETY

Flags

```
// Introduce class for typesafe flags
template <typename BitType, typename MaskType = VkFlags>
class Flags {
...
};

// BitType is scoped enum
enum class QueueFlagBits {
  eGraphics       = VK_QUEUE_GRAPHICS_BIT,
  eCompute        = VK_QUEUE_COMPUTE_BIT,
  eTransfer       = VK_QUEUE_TRANSFER_BIT,
  eSparseBinding = VK_QUEUE_SPARSE_BINDING_BIT
};

// Define typesafe flags
typedef Flags<QueueFlagBits, VkQueueFlags> QueueFlags;
```

TYPE SAFETY

Flags

Examples:

```
vk::QueueFlags bits1; // create flags with no bits set

vk::QueueFlags bits2 = vk::QueueFlagBits::eGraphics;

vk::QueueFlags bits3 = vk::QueueFlagBits::eGraphics | vk::QueueFlagBits::eCompute;

bits3 |= vk::QueueFlagBits::eTransfer;

if (bits3 & vk::QueueFlagBits::eGraphics) {}

if (!(bits3 & vk::QueueFlagBits::eGraphics)) {}

someFunctionWithQueueFlags({}); // pass {} for flags with no bit set.

bits3 &= ~vk::QueueFlagBits::eTransfer; // not yet implemented, emulate with | bit, ^ bit
```

INITIALIZATION

CreateInfos and Structs

```
class EventCreateInfo
{
public:
    // All constructors initialize sType/pNext
    EventCreateInfo(); // Initialize all fields with default values (0 currently)
    EventCreateInfo(EventCreateFlags flags); // Create with all parameters specified
    EventCreateInfo(VkEventCreateInfo const & rhs); // Construct from native Vulkan type

    // get parameter object
    const EventCreateFlags& flags() const;
    EventCreateFlags& flags(); // get parameter from non-cost object

    // set parameter
    EventCreateInfo& flags(EventCreateFlags flags);
    ...
    operator const VkEventCreateInfo&() const; // cast operator to native vulkan type
};
```

RESULTS

CODE SIZE & TYPE SAFETY

```
VkApplicationInfo appInfo = {};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pNext = NULL;
appInfo.pApplicationName = appName;
appInfo.applicationVersion = 1;
appInfo.pEngineName = engineName;
appInfo.engineVersion = 1;
appInfo.apiVersion = VK_MAKE_VERSION(1, 0, 30);

VkInstanceCreateInfo i = {};
i.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
i.pNext = NULL;
i.flags = 0;
i.pApplicationInfo = &appInfo;
i.enabledLayerCount = layerNames.size();
i.ppEnabledLayerNames = layerNames.data()
i.enabledExtensionCount = extNames.size();
i.ppEnabledExtensionNames = extNames.data();

VkInstance instance;
VkResult res = vkCreateInstance(&i, NULL, &instance);
assert(res == VK_SUCCESS);
```

```
vk::ApplicationInfo appInfo(appName, 1,
                            engineName, 1,
                            VK_MAKE_VERSION(1,0,30));
vk::InstanceCreateInfo i({}, &appInfo,
                         layerNames.size(), layerNames.data(),
                         extNames.size(), extNames.data());
vk::Instance instance;
vk::Result res = vk::createInstance(&i, nullptr, &instance);
assert(res == vk::Result::eSuccess);
```

DESIGNATED INITIALIZER LIST

```
VkApplicationInfo appInfo =
{
  .sType = VK_STRUCTURE_TYPE_APPLICATION_INFO,
  .pNext = NULL,
  .pApplicationName = appName,
  .applicationVersion = 1,
  .pEngineName = engineName,
  .engineVersion = 1,
  .apiVersion = VK_MAKE_VERSION(1,0,30)
};
```

```
vk::ApplicationInfo appInfo = vk::ApplicationInfo()
  .setPApplicationName(appName)
  .setAppplicationVersion(1)
  .setPEngineName(engineName)
  .setEngineVersion(1)
  .setApiVersion(VK_MAKE_VERSION(1,0,30));
```

Names explicit, but no guarantee to set all fields

Designated initializer list not part of C++1X

C++ CODING STYLE

HANDLES

```
VkCommandBuffer cmd = ...
VkRect2D scissor;
scissor.offset.x = 0;
scissor.offset.y = 0;
scissor.extent.width = width;
scissor.extent.height = height;
vkCmdSetScissor(cmd, 0, 1, &scissor);
```

Convert C-Style OO
to
C++ Style OO

```
vk::CommandBuffer cmd;
cmd.setScissor(0, 1, &scissor);
```

```
class CommandBuffer
{
    // conversion from/to native handle
    CommandBuffer(VkCommandBuffer commandBuffer);
    CommandBuffer& operator=(VkCommandBuffer
                             commandBuffer);
    operator VkCommandBuffer() const;

    // boolean tests if handle is valid
    explicit operator bool() const;
    bool operator!() const;

    // functions
    void setScissor(uint32_t firstScissor,
                    uint32_t scissorCount,
                    const Rect2D* pScissors) const;

};
```

C++ CODING STYLE

TEMPORARY STRUCTS, RETURN VALUES & ERROR HANDLING

```
class Device {
  Result createFence(const FenceCreateInfo * createInfo,
                     AllocationCallbacks const * allocator, Fence * fence) const;
};
```

Change from pointer to reference
allows passing temporaries

```
class Device {
  ResultValueType<Fence>::type createFence(const FenceCreateInfo & createInfo,
                     Optional<AllocationCallbacks const> const & allocator) const;
};
```

C++ CODING STYLE

TEMPORARY STRUCTS AND EXCEPTIONS

```
class Device {
  Result createFence(const FenceCreateInfo * createInfo,
                     AllocationCallbacks const * allocator, Fence * fence) const;
};
```

AllocationCallbacks are optional and might be null
Optional<Allocationcallbacks> accepts nullptr as input

```
class Device {
  ResultValueType<Fence>::type createFence(const FenceCreateInfo & createInfo,
                     Optional<AllocationCallbacks const> const & allocator) const;
};
```

C++ CODING STYLE

Optional References and Return Values

```
class Device {
  Result createFence(const FenceCreateInfo * createInfo,
                     const AllocationCallbacks * allocator, Fence * fence) const;
};
```

Fence is now a return value

```
class Device {
  ResultValueType<Fence>::type createFence(const FenceCreateInfo & createInfo,
                     Optional<AllocationCallbacks const> const & allocator) const;
};
```

C++ CODING STYLE

ResultValue<T>::type

Why not return just T instead of ResultValueType<T>::type?

Some people can‘t or won‘t enable exceptions -> define VULKAN_HPP_NO_EXCEPTIONS

In this case ResultValueType<T>::type is

- `void` if the function either have a return value nor a result
- `T` if the function does not have a return value
- `ResultValue<T>` in all other cases. This struct holds result/value
  - has std::tie(result, value) support!

C++ CODING STYLE

RESULTS

```
vk::Fence fence;
vk::FenceCreateInfo ci;
vk::Result result = device.createFence(&ci, nullptr, &fence);
assert(result == vk::Result::eSuccess);
```

```
try {
  vk::Fence fence = device.createFence({}, nullptr);
} catch (std::system_error e) {...}
```

C++ CODING STYLE

RESULTS

```
vk::Fence fence;
vk::FenceCreateInfo ci;
vk::Result result = device.createFence(&ci, nullptr, &fence);
assert(result == vk::Result::eSuccess);
```

```
vk::Fence fence = device.createFence({}, nullptr);
```

C++ CODING STYLE

ARRAYS AS PARAMETER INPUT

```
VkCommandBuffer cmd = ...
VkRect2D scissor;
scissor.offset.x = 0;
scissor.offset.y = 0;
scissor.extent.width = width;
scissor.extent.height = height;
cmd.setScissor(0, 1, &scissor);
```

(count, ptr)

```
class CommandBuffer
{
  // additional functions for arrays
  void setScissor(uint32_t firstScissor,
                  ArrayProxy<Rect2D> const & scissors)
                  const;

};
```

```
VkCommandBuffer cmd = ...
cmd.setScissor(0, { { {0u,0u}, {width, height} } });
```

Offset2D Extent2D

Rect2D

std::vector<Rect2D>

Count and real data size always correct
Lifetime of temporary is function call

C++ CODING STYLE

ARRAYS

`vk::ArrayProxy<T>` abstracts passing arrays of data to Vulkan

```
{} // empty list, count = 0

{nullptr} // nullptr results in count = 0

{rect2D} // Single element, count = 1

{ptr, count} // ptr, count pair

{ {rect1, rect2, rect3} } // std::initializer_list -> no overhead if temporary

std::vector<T, allocator> // dynamic lists

std::array<size, T> // staticically sized arrays
```

C++ CODING STYLE

Enumerations / Queries

```
vk::PhysicalDevice pd = ...;
std::vector<vk::QueueFamilyProperties> properties;
uint32_t queueCount;
```

Define variables

```
pd.getQueueFamilyProperties(&queueCount, nullptr);
```

Query size

```
properties.resize(queueCount);
```

Resize std::vector

```
pd.getQueueFamilyProperties(&queueCount, properties.data());
```

Query data

Short version:

```
vk::PhysicalDevice pd = ...;
std::vector<vk::QueueFamilyProperties> properties = pd.getQueueFamilyProperties();
```

C++ CODING STYLE

Dynamically sized queries

```
vk::Result res;
std::vector<vk::ExtensionProperties> properties;
uint32_t count;
char *layer = ...;
do {
  res = vk::enumerateInstanceExtensionProperties(layer, &count, nullptr);
  if (res) throw error;

  properties.resize(count);
  res = vk::enumerateInstanceExtensionProperties(layer, &count, properties.data());
} while (res == vk::Result::eIncomplete);
```

```
std::vector<vk::ExtensionProperties> properties = enumerateInstanceExtensionProperties(layer);
```

C++ CODING STYLE

updateBuffer

```
class CommandBuffer {
    void updateBuffer(Buffer dstBuffer, DeviceSize dstOffset,
                      DeviceSize dataSize, const uint32_t* pData) const;
};
```

dataSize in bytes

data type is uint32_t -> dataSize must be multiple of 4

```
cmd.updateBuffer(buffer, 0,
                 static_cast<DeviceSize>(data.size() * sizeof(*data.data())),
                 reinterpret_cast<const uint32_t*>(data.data()));
```

Might work or might not work if `sizeof(*data.data())` is not a multiple of 4

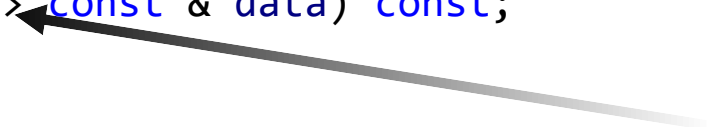
Call is not that beautiful

C++ CODING STYLE

updateBuffer

```
class CommandBuffer {
    template <typename T>
    void updateBuffer(Buffer dstBuffer, DeviceSize dstOffset,
                      std::vector<T> const & data) const;
};

cmd.updateBuffer(buffer, 0, data);
```



static_assert to ensure sizeof(T) is a multiple of 4
-> compile time failure instead of runtime failure

UTILITY FUNCTIONS

to_string

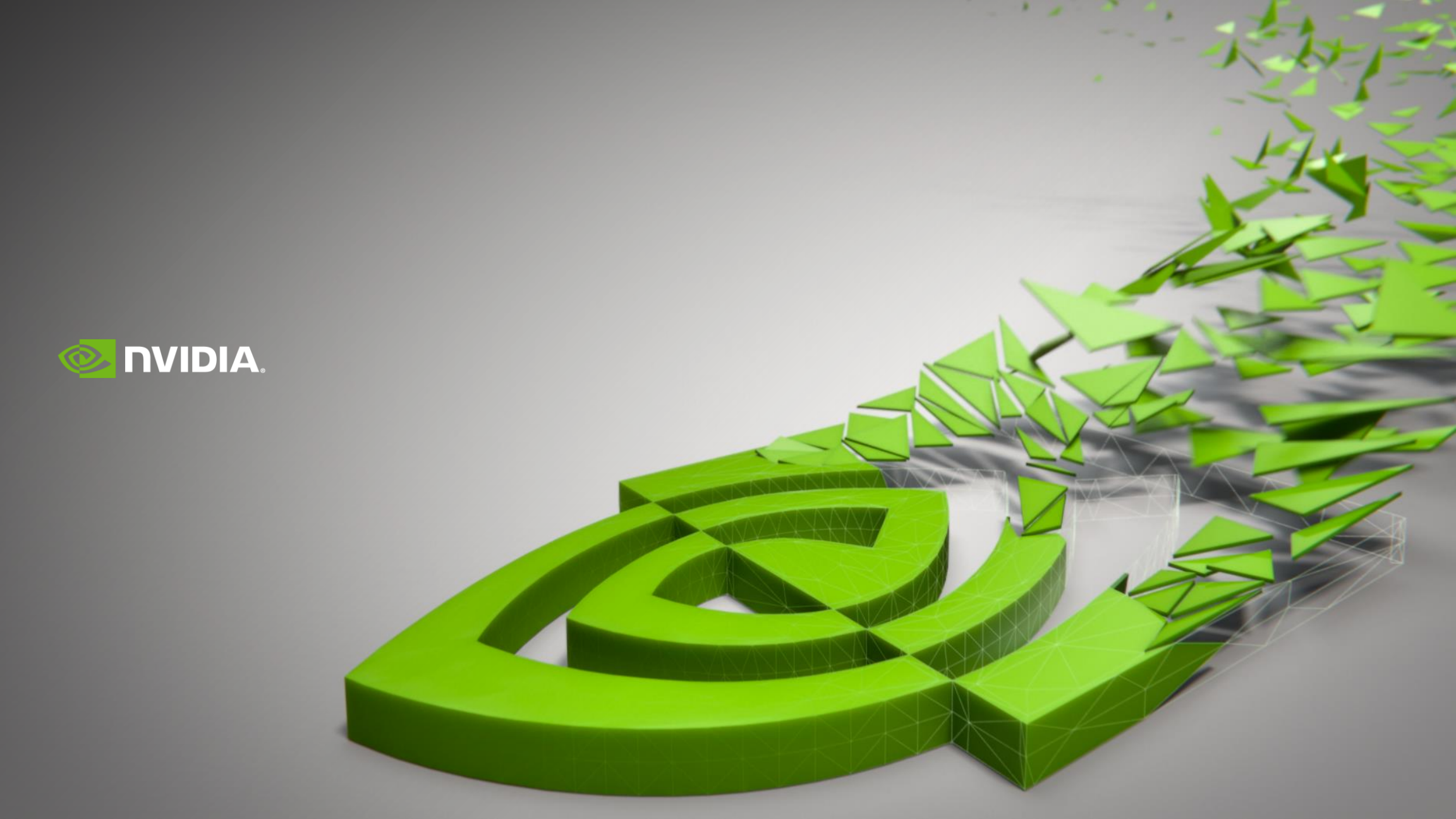
For debugging purposes it‘s useful to convert enums or flags to strings

```
std::string to_string(FooEnum value)  for enums

to_string(vk::Result::eSuccess)
-> “Success”


std::string to_string(BarFlags value) for flags

to_string(vk::QueueFlagBits::eGraphics | vk::QueueFlagBits::eCompute)
-> “Graphics | Compute”
```


NVIDIA