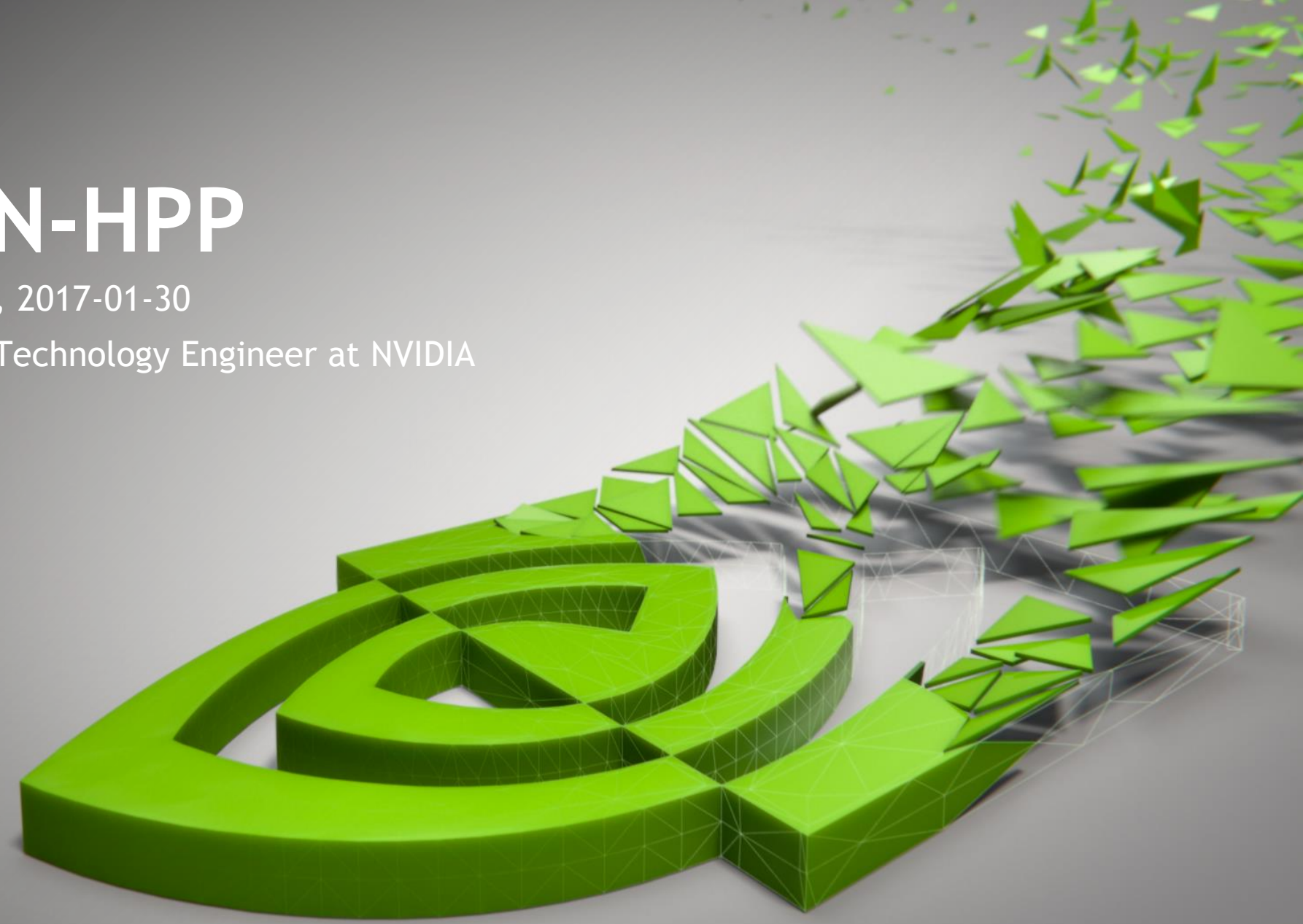
VULKAN-HPP

Markus Tavenrath, 2017-01-30

Senior Developer Technology Engineer at NVIDIA



WHAT IS VULKAN-HPP

Vulkan-Hpp is an autogenerated C++11 binding for Vulkan

Goal: Integrate C++11 features as seamless as possible into Vulkan

1. Improve error detection at compile time
2. Reduce amount of code to type
3. Simplify runtime error handling
4. No additional runtime overhead

VULKAN NAMESPACE

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

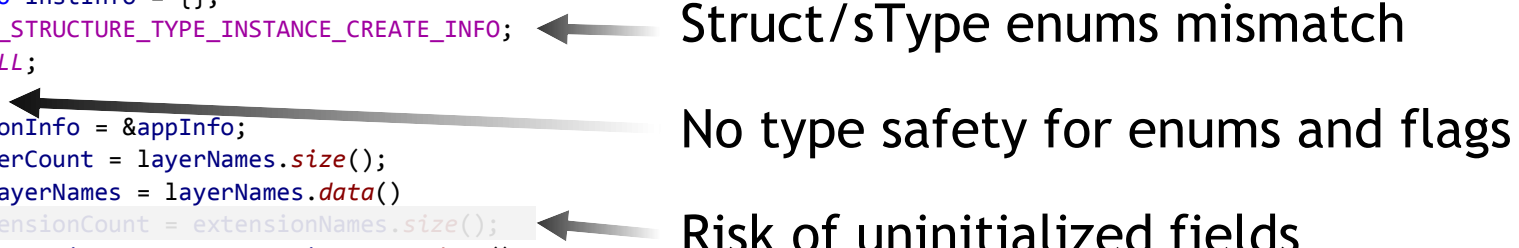
avoid symbol collisions

```
// 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
};

// Alias for Flags
using QueueFlags = Flags<QueueFlagBits, VkQueueFlags>;
```

TYPE SAFETY

Flags

Examples:

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

vk::QueueFlags bits2 = vk::QueueFlagBits::eGraphics; // assign bits directly

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

bits3 |= vk::QueueFlagBits::eTransfer; // compound assignment operator

if (bits3 & vk::QueueFlagBits::eGraphics) {} // test if bit is test

if (!(bits3 & vk::QueueFlagBits::eGraphics)) {} // test it bit is not test

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

bits3 &= ~vk::QueueFlagBits::eTransfer; // applies ~ only bits supported in FlagBits
```

INITIALIZATION

CreateInfos and Structs

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

    // Emulate designated initializer pattern: ci.setFoo(foo).setBar(bar);
    EventCreateInfo& setFlags(EventCreateFlags flags);
    ...

    operator const VkEventCreateInfo&() const; // cast operator to native vulkan type

    // Direct access to all fields to make porting to C++11 bindings easy.
    EventCreateFlags flags;
};
```

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, 39);

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(1,0,39));
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);
```

C++ 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 C-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;
};
```

HANDLES

For 32-bit compilers non-dispatchable Vulkan handles are not typesafe

```
#define VK_DEFINE_NON_DISPATCHABLE_HANDLE(object) typedef uint64_t object;
```

Explicit cast required: `vk::Device = static_cast<vk::Device>(c_device)`

Tell the compiler that you know what you‘re doing:

```
#define VULKAN_HPP_TYPESAFE_CONVERSION 1
```

to take the risk of implicit `VkHandle` -> `vk::Handle` conversion

PARAMETER HANDLING

TEMPORARY STRUCTS, RETURN VALUES & EXCEPTIONS

```
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;
};
```

PARAMETER HANDLING

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;
};
```

PARAMETER HANDLING

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;
};
```

PARAMETER HANDLING

ResultValue<T>::type

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

Exceptions can be disabled: `#define VULKAN_HPP_NO_EXCEPTIONS 1`

In this case `ResultValueType<Fence>::type` is

- `void` if the function neither has a return value nor a result
- `Fence` if the function does not have a return value of type `Vk::Result`
- `ResultValue<Fence>` in all other cases
  - This struct holds a `Vk::Result` and `Vk::Fence`
  - It also supports `std::tie(result, value)` for clean syntax
  - C++17: `auto [result, value] = device.createFence(…);`

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) {...}
```

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);
```

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::initialiser_list<Rect2D>

Count and real data size always do match

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 additional cost

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

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

ENUMERATIONS & QUERIES

C-API

```
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);
```

C++ API

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

UTILITY FUNTIONS

vk::UniqueHandle

RAII is used in several Vulkan C++ libraries and frameworks.

People like it, so we added `vk::UniqueHandle<ObjectType, Deleter>`

```
vk::UniqueDevice device = pdev.createDeviceUnique(…);

vk::UniqueFence fence = device->createFenceUnique({});

device->waitForFences( *fence, false, timeout );
```

Handle gets destructed automatically once `vk::UniqueHandle` goes out of scope

Not for free! Each `vk::UniqueHandle` has to store `parent` and `allocator`

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”
```

Questions?

https://github.com/KhronosGroup/Vulkan-Hpp