Vulkan®

DEVELOPER DAY

Vulkan-Hpp

Markus Tavenrath, NVIDIA

KHRONOS™ GROUP

UBISOFT

MONTRÉAL
APRIL 2018

What is Vulkan-Hpp

- **C++ bindings to Vulkan**
- **Fully generated from Vulkan-Spec**
- **Keep the spirit of Vulkan**
- **Map concepts of C API to C++ features at no to minimal runtime cost**
  - Type safety for enums, flags and handles
  - C++ class handles which member functions
  - Error handling through exceptions (optional)
  - Support for C++ arrays in function calls
  - Easy enumerations/queries
  - Extension loading
  - A UniqueHandle

Vulkan-Hpp namespace

- **Vulkan-Hpp lives in ‚vk‘ namespace**
  - Plan to move it to khr::vk namespace for easier use
  - Engines have engine::vk namespace -> creates ambiguity for symbols
- **Namespace has all Vulkan core symbols**
  - Functions, structs and handles have ‚vk‘ prefix removed
    - vkCreateImage gets vk::createImage (function)
    - vkImageTiling gets vk::ImageTiling (enum)
    - vkImageCreateInfo gets vk::ImageCreateInfo (struct)

Enums

- Enums are implemented as scoped enums
- Gives us type safety at compile time
  - Not a frequent error, but might be hard to find. Validation might find it or not
- Access them through `EnumType::eEnumName`, e.g. `ImageViewType::e1D`
- The e prefix is unfortunate
  - Least evil of all solution to solve ‚symbol starts with digit‘ problem

```
enum class ImageViewType
{
    e1D = VK_IMAGE_VIEW_TYPE_1D,
    e2D = VK_IMAGE_VIEW_TYPE_2D,
    e3D = VK_IMAGE_VIEW_TYPE_3D,
    eCube = VK_IMAGE_VIEW_TYPE_CUBE,
…
};
```

Handles

- Each handle is a unique class and thus typesafe
- Initialized initialize `VK_NULL_HANDLE`
- Interop with C-API is possible
  - Implicit cast enabled on 64-bit targets
  - explicit cast required on 32-bit targets because Vulkan handles are typed `uint64_t`
- Member function for each Vulkan function accepting handle as first parameter
  - `vkCreateImage(device, ...) -> device.createImage(...)`

Arrays

- Vulkan accepts arrays as (ptr, count) pair
- Pair defined in spec, so easy to identify
- C++ code uses classes for arrays in a lot of cases
- Vulkan-Hpp defines ArrayProxy class which accepts all important standard types
    - `nullptr_t` (empty array)
    - A single value (array of size 1)
    - `(count, ptr)` pair
    - `std::initializer_list`
    - `std::array`
    - `std::vector`
- Each member function accepting an array will accept ArrayProxy class
- Temporaries supported

ArrayProxy examples 1

```
vk::CommandBuffer c;

// pass an empty array
c.setScissor(0, nullptr);

// pass a single value. Value is passed as reference
vk::Rect2D scissorRect = { { 0, 0 },{ 640, 480 } };
c.setScissor(0, scissorRect);

// pass a temporary value.
c.setScissor(0, { { 0, 0 },{ 640, 480 } });
```

ArrayProxy examples 2

```
// Put std::initializer_list on stack and pass it
vk::Rect2D scissorRect1 = { { 0, 0 },{ 320, 240 } };
vk::Rect2D scissorRect2 = { { 320, 240 },{ 320, 240 } };
c.setScissor(0, { scissorRect, scissorRect2 });

// Pass temporary std::initializer_list
c.setScissor(0, { { { 0,   0 },{ 320, 240 } },
  { { 320, 240 },{ 320, 240 } }
  }
);
```

ArrayProxy examples 3

```
// pass a std::array
std::array<vk::Rect2D, 2> arr{ scissorRect1, scissorRect2 };
c.setScissor(0, arr);

// pass a std::vector of dynamic size
std::vector<vk::Rect2D> vec;
vec.push_back(scissorRect1);
vec.push_back(scissorRect2);
c.setScissor(0, vec);
```

ArrayProxy custom type

- Custom array classes are supported too
- Implement implicit cast operator to vk::ArrayProxy in your class

```
class FooArray {
public:
  operator vk::ArrayProxy() const {
    return vk::ArrayProxy(count, ptr);
  }
 …
};
```

CreateInfo structs

- CreateInfo structs initialize `sType` and `pNext fields`
  - No wrong sType by accident
  - pNext always nullptr_t to avoid crashes
- Constructors of CreateInfo structs accept all field members
  - Allows temporaries and more compact code if desired

```
vk::Image image = device.createImage({
  {}, vk::ImageType::e2D, vk::format::eR8G8B8A8Unorm,
  { width, height, 1 },
  1, 1, vk::SampleCount::e1,
  vk::ImageTiling::eOptimal, vk::ImageUsage:eColorAttachment,
  vk::SharingMode::eExclusive, 0, 0, vk::Imagelayout::eUndefined }
);
```

Structure pointer chains

- Vulkan provides `pNext` mechanism to link `CreateInfo` structs.
- Vulkan spec specifis valid pNext objects for each `CreateInfo` struct.
- `pNext` is declared as `void*`. Can‘t check directly.
- Vulkan-Hpp provides validation class with storage to verify links at compile time

```
// Create a structure pointer chain
vk::StructureChain<vk::MemoryAllocateInfo, vk::ImportMemoryFdInfoKHR> c;

// Fetch components of pointer chain
vk::MemoryAllocateInfo &allocInfo = c.get<vk::MemoryAllocateInfo>();
vk::ImportMemoryFdInfoKHR &fdInfo = c.get<vk::ImportMemoryFdInfoKHR>();

// Use &c or &allocInfo as pNext
```

Enumerations in Vulkan

- Enumeration logic complex

```
std::vector<LayerProperties, Allocator> properties;
uint32_t propertyCount;
Result result;
do
{
  // determine number of elements to query
  result = static_cast<Result>(vk::enumerateDeviceLayerProperties(m_physicalDevice, &propertyCount, nullptr));
  if ((result == Result::eSuccess) && propertyCount)
  {
    // allocate memory & query again
    properties.resize(propertyCount);
    result = static_cast<Result>(vk::enumerateDeviceLayerProperties(m_physicalDevice, &propertyCount, reinterpret_cast
      <VkLayerProperties*>(properties.data())));
  }
} while (result == Result::eIncomplete); // it's possible that the count has changed, start again if properties was not big enough
// resize to real property count, might have shrunk
properties.resize(propertyCount);
```

Enumerations in Vulkan-Hpp

- Vulkan-Hpp identifies query and enumerations and generates simple call

  -
    ```
    std::vector<LayerProperties> properties =
    physicalDevice.enumerateDeviceLayerProperties();
    ```

- Custom allocator is supported too

  -
    ```
    std::vector<LayerProperties, FooAllocator> properties =
    physicalDevice.enumerateDeviceLayerProperties<FooAllocator>();
    ```

Exceptions & Return value conversion

- Optional feature, original version of function supported too
- Functions which return values usually return errors
- Without exceptoins:

```
vk::Buffer buffer;
vk::Result result = device.createBuffer(..., &buffer);
if (result != VK_SUCCESS) { // error handling}
// repeat code from above multiple times
```

- With exceptions:

```
vk::Buffer buffer;
try {
  buffer = device.createBuffer(...);
  // ... create more handles here
} catch(...) {
  clean up at single location;
}
```

UniqueHandle

- If code is not in critical performance section UniqueHandle is a nice helper class
- Works like std::unique_ptr
-
  ```
  try {
    vk::UniqueHandle<vk::Buffer> buffer = device.createBuffer(...);
    ... create more handles here
  } catch(...) {
      // no need to cleanup here
  }
  ```

Extension loading

- Vulkan-Hpp relies on all Vulkan available by default
- Loader does not export all of them, dispatch table is required
- Vulkan-Hpp allows passing a dispatch table as last parameter

```
// This dispatch class will fetch all function pointers through the
passed instance
vk::DispatchLoaderDynamic dldi(instance);

// This dispatch class will fetch function pointers for the passed
device if possible, else for the passed instance
vk::DispatchLoaderDynamic dldid(instance, device);

// Pass dispatch class to function call as last parameter
device.getQueue(graphics_queue_family_index, 0, &graphics_queue, dldid);
```

Samples / Ecosystem

- People are asking for samples frequently
- Started to develop a few samples in the Vulkan-Hpp samples subdirectory!
- If you wrote a Vulkan-Hpp sample do a PR
- If you use Vulkan-Hpp we‘re happy to add you to the README

Questions?

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

matavenrath@nvidia.com