Auto-VK-Toolkit - C++20 Modules

Hamed Jafari Sahamieh

May 2023

Introduction

The purpose of this project is described at https://github.com/cg-tuwien/Auto-Vk-Toolkit/issues/95 in detail.

Findings

As described by the aforementioned link, C++20 Modules do have the potential to significantly improve compile time. However, the combination of their current implementation state, maturity and compatibility makes their usage difficult, especially in combination with heavy, external headers.

Externals actually have a somewhat crucial position in this project. Libraries such as `Vulkan` and `GLM` (among others) are used across many areas of the framework. They are also large and not undemanding to recompile. This is the source of the problem somewhat: If the pre-compiled header file (PCH) is updated in any form, all external headers have to be recompiled. Another aspect is that in general PCHs and C++ modules do not necessarily function well together in a project. Module files cannot access PCHs. This has large precautions as in many cases PCHs cannot be used as fall-backs where modules are not possible. For instance if `vulkan.hpp` is to be compiled as part of a PCH, then no module in the project can access Vulkan entities, unless a second copy is made (for each module).

In this regard ensuring that these headers can be transferred to modules is a priority. Unfortunately this is not an easy step. In this context it is important to distinguish between named modules and header units, see: https://learn.microsoft.com/en-us/cpp/build/compare-inclusion-methods.

Attempting to compile `auto_vk_toolkit.hpp`, or `avk.hpp`, or anything of that scale as header unit flat out fails. They are also too large and complicated to directly be turned into modules (and also not the point as this would just shift the issue from PCHs to modules/header units). For the purpose of testing whether these large externals can be turned into their own translation units we focused on Auto-Vk code and tried to compile `vulkan.hpp` separately at this level. This means that `avk.hpp` was first divided into many smaller parts and the various headers were disentangled. Major parts of Auto-Vk were then turned into C++20 modules that rely on a main Vulkan module or header unit yet to be created. Then we attempted to recompile these headers with `vulkan.hpp` as its own translation units.

Focusing on header units first, one that contains `vulkan.hpp` could look as follows:

```
#pragma once

//vulkan_header_unit.hpp

[...]
#define Vulkan_HPP_ENABLE_DYNAMIC_LOADER_TOOL 0
#define VK_ENABLE_BETA_EXTENSIONS
#include <vulkan/vulkan.hpp>
```

This header is then compiled as a header unit in visual studio and then accessed by using `import "vulkan_header_unit.hpp";`. Attempting to access the functionality offered by Vulkan will eventually lead to internal compiler errors. Further investigations show that certain types cannot be found.

For example, the MSVC compiler seemed to to have issues with a variant type containing `vk::AccessFlags2KHR` in combination with `std::get`. Suddenly the problem is solved if the following code snippet is included in the header unit:

```
//test
using abc = std::variant<std::monostate, vk::AccessFlags2KHR, uint8_t>;

vk::AccessFlags2KHR& getAccess(abc& a) {
    return std::get<1>(a);
}

const vk::AccessFlags2KHR& getAccess(const abc& a) {
    return std::get<1>(a);
}
```

The header units documentation by Microsoft[1] states that the method can be used for "well behaving headers". After many experiments we assume that headers such as `Vulkan` and `GLM` are not necessarily "well behaved".

Another approach is to use named modules. It is important to mention that the following approach to export an entire interface does not seem to work (at least with Visual Studio):

[1]Compare header units, modules, and precompiled headers: https://learn.microsoft.com/en-us/cpp/build/compare-inclusion-methods?view=msvc-170

```
module; //vulkan_module.ixx

#define Vulkan_HPP_NAMESPACE vk
#define Vulkan_HPP_DISPATCH_LOADER_DYNAMIC 1
#include <vulkan/vulkan.hpp>

export module Vulkanmodule;

export {
    using namespace vk;
}
```

In this case the namespace and its content cannot be found by the files importing the module. So the following was attempted:

```
module;  //vulkan_module.ixx

#define Vulkan_HPP_NAMESPACE vkbase
#define Vulkan_HPP_DISPATCH_LOADER_DYNAMIC 1
#include <vulkan/vulkan.hpp>

export module Vulkanmodule;

export namespace vk {
    using PhysicalDevice = vkbase::PhysicalDevice;
    using PhysicalDeviceMemoryProperties
        = vkbase::PhysicalDeviceMemoryProperties;
    /* and so on for every class required... */
}
```

Down the line, this resulted in errors similar to the ones found above with header units. Incomplete types or missing methods.

More research revealed that this may have to do with the 'discarding' behaviour of the compiler. The following link describes the problem: https://vector-of-bool.github.io/2019/10/07/modules-3.html. This also explains why including miss-behaving files (such as `vk::AccessFlags2KHR` above) in the definitions solves the problem. However, this cannot really be considered a robust solution.

The conclusion from the above experiments is that `vulkan.hpp` (and possible other externals of that complexity) is not a suitable candidate for C++20 modules, without a major rework. Similar issues were raised with `GLM`, although less time was spend there in trying to remedy the situation.

Other Notes

- Any item that has internal linkage will not be exported by modules, and it can be difficult to spot such issues (for instance static functions) when doing a mass export: `export namespace avk [...]`. It is best to avoid internal linkage[2] if the project is to be ultimately converted.
- The current implementation of modules is still not bug free or smooth. As far as observed on Visual Studio forums and bug reports, issues were still being resolved as of end 2022 and beginning of 2023. Issues ranged from excessive and erroneous compiler failures and warnings to problems with IDE itself.
- The C++20 modules are not supported fully by all compilers.
- Features such as header units (as well as their implementation details) are very dependant on individual compilers.
- Modules are to be extended in C++23 (at least including standard library as module) and possibly beyond. Most discussions observed on the web do not recommend using modules at this stage for more complicated constructs.

All of the above being said, modules work just fine for those project headers which do not rely on external libraries. The issue however is that they do not bring much benefit due to their scale, and also limited the project by complicating PCH utilization.

Beyond Modules

It was concluded that a more flexible PCH setup is more beneficial. PCHs are close to optimal when recompiling is not triggered frequently. The following steps were taken:

- Minimize the presence of Auto-Vk-Toolkit headers in the PCH file. This obviously leads to less recompile triggers.
- Remove PCH from the files that do not need it. Circular PCH dependencies can easily cause unnoticed triggers. Also a recompilation of PCH leads to recompilation of those units that do not need recompiling.
- Clean the project tree as far as possible. Less complicate tree leads to less mistakes causing recompilation of units. There might be headers that have been missed in the process.
- As soon as Auto-Vk-Toolkit's interface becomes more stable, they can be moved back into the PCH. The current setup has a relatively high 'first time' compilation, but less external recompile triggers.

[2] Linkage types in C++: https://vector-of-bool.github.io/2019/10/07/modules-3.html

Suggestions

- Be aware that PCHs are compiled once per project. This means it may be helpful to reduce project count and/or combine functionality. Users should be encouraged to only compile those projects that they work with and not the entire solution to reduce compile time as this can be significant.

- Some headers (for instance `windows.hpp` or context headers) are quite heavy and also present in many areas. However not all the functionality is needed. A more incremental design (more layers) may reduce the need of the including translation units to scan the entirety of these files. The (more stable) core of such headers can then be left in PCH. The context files (generic, Vulkan etc.) also could use a redesign, because they are quite inter-wined (saying this without knowing them very well).

- There is a major problem with the project setup: Most translation units take the property of whether they should use PCH from the project. Also there is a forced include (`cg_stdafx.hpp`). The problem is that even the files that do not need the PCH (see external folder with `FileWatcher` and so on) get recompiled when PCH changes. The reason apparently is that they need to redo the discarding process. Some of these were fixed however, further work may be necessary. Another suggestion is to remove the forced include and make PCH inclusion explicit.

- Using a proven logger library can make the project quite a bit less complicated, in terms of design (For instance `spdlog`).

- Reduce the number of cpp files where it makes sense.

- Incomplete type, external definitions: Designing translation units with incomplete types in headers (where possible) and more external definitions will definitely improve compile times. Basically the idea is to move more code away from header files.

- Another suggestion is to port `GLM` to a module as soon as some improvements are made to the whole system. This will potentially have a very large effect as `GLM` is used in many areas. This would mean 2 compilations of `GLM` (one for files that need Vulkan down the hierarchy), but may significantly reduce the dependency on the PCH.