Computers & Graphics 111 (2023) 155–165

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on EG2022 Edu Best Papers

Vulkan all the way: Transitioning to a modern low-level graphics API in academia

Johannes Unterguggenberger *, Bernhard Kerbl, Michael Wimmer

TU Wien, Institute of Visual Computing & Human-Centered Technology, Favoritenstr. 9-11/E193-02, 1040 Vienna, Austria

ARTICLE INFO

Article history:
Received 29 October 2022
Received in revised form 9 January 2023
Accepted 1 February 2023
Available online 9 February 2023

Keywords:
Real-time rendering
GPU
Graphics API
Teaching
Vulkan
Programming framework

ABSTRACT

For over two decades, the OpenGL API provided users with the means for implementing versatile, feature-rich, and portable real-time graphics applications. Consequently, it has been widely adopted by practitioners and educators alike and is deeply ingrained in many curricula that teach real-time graphics for higher education. Over the years, the architecture of graphics processing units (GPUs) incrementally diverged from OpenGL's conceptual design. The more recently introduced Vulkan API provides a more modern, fine-grained approach for interfacing with the GPU, which allows a high level of controllability and, thereby, deep insights into the inner workings of modern GPUs. This property makes the Vulkan API especially well suitable for teaching graphics programming in university education, where fundamental knowledge shall be conveyed. Hence, it stands to reason that educators who have their students' best interests at heart should provide them with corresponding lecture material. However, Vulkan is notoriously verbose and rather challenging for first-time users, thus transitioning to this new API bears a considerable risk of failing to achieve expected teaching goals. In this paper, we document our experiences after teaching Vulkan in both introductory and advanced graphics courses side-by-side with conventional OpenGL. A collection of surveys enables us to draw conclusions about perceived workload, difficulty, and students' acceptance of either approach. In doing so, we identify suitable conditions and recommendations for teaching Vulkan to both undergraduate and graduate students.

1. Introduction

For many years, OpenGL has remained the default choice for teaching undergraduate students the use of graphics APIs. Its high portability as well as an extensive body of documentation, guides, and tooling options (e.g., open-source software emulators) made it the logical choice for accommodating students from different curricula. However, there are clear indicators that we are at a juncture where teaching OpenGL to undergraduate students is no longer adequate: Its API design as a state machine is often considered bothersome and, in many cases, no longer reflects the underlying hardware architecture. More severely, several interesting and desirable features of modern APIs, such as push constants or hardware-accelerated ray tracing, are simply not supported by OpenGL. The practical reasons for and against using OpenGL today are succinctly illustrated by our own experience using it in research. In our work on fast multi-view rendering [1], we already felt the age of OpenGL. Its usage turned out to be more error-prone due to the lack of proper error messages when compared to the modern low-level graphics API of our choice: Vulkan [2]. For our work on computing and exploiting conservative meshlet bounds [3], we switched to Vulkan, since it abstracts the hardware on a lower level than OpenGL, offering more insights and much more fine-grained control over the actual work that is carried out by GPUs, leading to a better and more productive development experience once learned. Consequently, our goal was set towards making the transition from OpenGL to Vulkan also in teaching in academia. The positive aspects of Vulkan are appealing, save for the small qualifier "once learned", which is exactly the crux of the matter.

Before going into details, we should argue about why to select Vulkan and not one of the other major modern, low-level graphics APIs: DirectX 12 [4] and Metal [5]. While most modern graphics APIs are similarly aligned in terms of usage principles and their level of hardware abstraction, only Vulkan is usable across all major desktop operating systems and across device categories (albeit only through an intermediate layer [6] on Apple platforms). Furthermore, it is an open industry standard defined by the members of the Khronos group, which includes all major GPU manufacturers, operating system manufacturers, and other individual, academic, and industry members [7]. They all contribute to shape and maintain the Vulkan API, while DirectX

* Corresponding author.
E-mail addresses: junt@cg.tuwien.ac.at (J. Unterguggenberger), kerbl@cg.tuwien.ac.at (B. Kerbl), wimmer@cg.tuwien.ac.at (M. Wimmer).

https://doi.org/10.1016/j.cag.2023.02.001

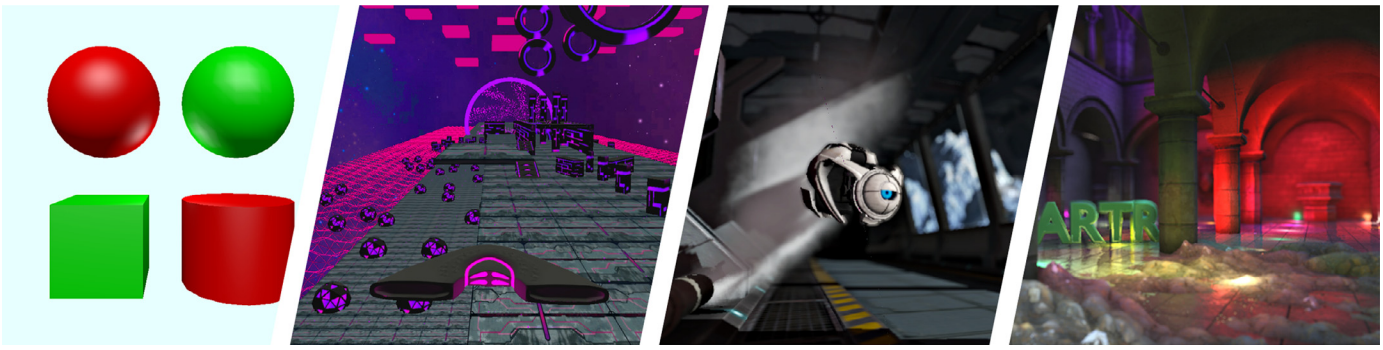


Fig. 1. Screenshots of implementations performed in our courses *Introduction to Computer Graphics*, *Computer Graphics Exercise*, *Real-Time Rendering*, and *Algorithms for Real-Time Rendering* (from left to right).

and Metal are proprietary standards, defined and controlled by a single company each. Vulkan appears to be not only the most future-proof API, but thanks to vendor-specific extensions, new hardware features are accessible in a timely manner. Hardware-accelerated ray tracing, for example, was available through an NVIDIA-specific extension [8] only one month after its availability in DirectX and has later been standardized [9]. Given these conditions, Vulkan is the sensible choice in higher education in our opinion.

The challenge of learning Vulkan is revealed when comparing source code and descriptive text for two of the most famous tutorials for drawing a single triangle to the screen: The OpenGL tutorial at *LearnOpenGL.com* requires fewer than 150 lines of code (LOC) on the host side [10]. In contrast, the de facto entry point for learning Vulkan at *vulkan-tutorial.com* ends up with approximately 700 LOC for achieving the same task and requires a much more extensive description for explaining the necessary setup leading up to this point [11]. The tutorials illustrate how Vulkan is indeed an API that operates on a much lower abstraction level than OpenGL. This implies that there are many more factors and talking points with Vulkan that must be addressed (at least to some degree) if taught to students. On the other hand, a potential upside thereof is that students receive a more fundamental knowledge about the inner workings of a modern GPU—and conveying fundamental knowledge constitutes a primary goal of higher education.

Computer science educators are in a position where they may struggle to identify a clear path for teaching Vulkan effectively. In many cases, an established course exists in the curriculum that relies on older, higher-level APIs. The big challenge then becomes incorporating the introduction of Vulkan and facilitating the transition to this new API for students, educators, and teaching assistants. This is a delicate maneuver since a hasty transition could disrupt and overwhelm each of these groups. In our case, we pondered different strategies: adding an entirely new course (which would imply a whole series of changes to subsequent courses in the curriculum), sticking to OpenGL in undergraduate and switching to Vulkan in graduate programs, or trying to introduce Vulkan as early as possible as an alternative to OpenGL. We ultimately decided to go with the latter, as it turned out to be minimally invasive curriculum-wise. Furthermore, we argue that learning Vulkan early on constitutes the highest benefit for students. After having used Vulkan for selected assignments of advanced courses since 2020, we successfully transitioned our introductory graphics course in 2021 into a double-tracked mode: We offered students the choice to stick with battle-tested OpenGL or embrace the new and potentially effortful Vulkan route.

We can conclude that supporting a Vulkan route was much less bumpy for our students than we initially anticipated, and therefore, we propose a pragmatic route for transitioning to Vulkan in academia for the purpose of teaching real-time computer graphics. In this paper, we describe the changes that we have made to transition an introductory graphics course to Vulkan in detail, and how we manage to keep workload in manageable bounds in an advanced graphics course that exclusively uses Vulkan. Furthermore, we share some statistics and experiences from several graphics courses where students can or must use Vulkan, providing detailed insights from our students' perspectives based on a set of questionnaires. Fig. 1 shows some representative results of student work that have been created in the different courses in our curriculum. This paper extends our previous work [12] with information about our meanwhile open-sourced framework *Vulkan Launchpad [13]*, description of the whole course structure of our real-time rendering education through bachelor and master programs, details about our Vulkan API usage in higher-level courses along with student questionnaire results about it, as well as information about the Vulkan frameworks used in higher-level courses: *Auto-Vk [14]* and *Auto-Vk-Toolkit [15]*.

2. Related work

Fundamental difficulties of students learning computer graphics and potential countermeasures are described by Suselo et al. [16]. We consider the difficulties with respect to mathematics, transforms and projections, and logical problem solving as preliminary challenges to learning graphics programming. Introducing graphics programming in an API-free manner is proposed by Chen et al. [17], which we see as an interesting pathway of education before learning how to access graphics hardware through a modern industry-standard API.

A possible syllabus for an introductory computer graphics course was described by Fink et al. [18]. Even though their concept of creating a comprehensive software-based rasterization framework for teaching graphics programming concepts in a more abstract and focused way is intriguing, the low-level aspects, which are crucial in real-time rendering, are hidden away from students. Students can obtain valuable insights from using an industry-standard graphics API, and comprehensive documentation and literature can be expected to be available for it in contrast to a proprietary framework. Furthermore, a custom software-based rasterization framework might be at high risk of diverging too much from developments in the industry or causing high maintenance efforts.

Based on the analysis performed by Balreira et al. [19], it can be concluded that OpenGL was the most widely used graphics API in university education in 2017, given the absence of any mention of other graphics APIs. We consider our suggestions and experiences described in this paper as being potentially relevant to every department that is thinking about migrating from teaching

OpenGL to teaching Vulkan but also to those who have already migrated. Experiences with the transition from legacy OpenGL to modern OpenGL in university education are described by Reina et al. [20]. They point to increased learning efforts in modern OpenGL due to reduced out-of-the-box convenience compared to legacy OpenGL. A similar point could be made when comparing Vulkan to modern OpenGL.

While Vulkan may provide a reasonable learning curve for developers who are proficient with various other APIs, it is notoriously difficult for students without prior experience. To fully appreciate Vulkan, users require an in-depth understanding of the underlying GPU hardware. The fine-grained control over work generation and scheduling necessarily make Vulkan verbose. Hence, students are confronted with the task of implementing a significant amount of boilerplate code for leveraging hardware features they may not yet fully understand. This situation is further aggravated by the absence of in-depth teaching material for Vulkan: comprehensive, thoroughly researched hands-on guide books, such as OpenGL's famed "Red Book" [21] or the "OpenGL Superbible" [22] are not yet available for Vulkan. Early attempts to provide additional abstraction or simplify the interface had only limited success [23]. However, Vulkan as an API is still evolving. Recent additions to the SDK, such as the `VK_KHR_dynamic_rendering` and `VK_KHR_synchronization2` extensions [24], aim to alleviate neuralgic pressure points of the API.

3. Course overview and details

Five courses represent the pathway in our visual computing curriculum from learning real-time graphics programming to mastering it. In the first course, no graphics API is used. In the next three, students can opt to use the Vulkan API. In our most advanced course, Vulkan use is mandatory. Our curriculum recommends to take the courses in the following order:

1. *Introduction to Visual Computing*
2. *Introduction to Computer Graphics*
3. *Computer Graphics Exercise*
4. *Real-Time Rendering*
5. *Algorithms for Real-Time Rendering*

The first three courses target bachelor/undergraduate students. *Introduction to Visual Computing* introduces rasterization and several other fundamental concepts of real-time rendering. Students are tasked with manually implementing selected parts of a rasterization pipeline, such as polygon clipping and line rasterization, but we refrain from using a graphics API for any of its tasks. *Introduction to Computer Graphics* is our students' first encounter with a graphics API, where buffers have to be prepared for GPU usage, commands have to be submitted to a GPU, and also shader programs are to be implemented. These skills can be put to practical use during the *Computer Graphics Exercise*, where students are tasked with programming a game from scratch based on a self-written game engine and self-implemented graphics effects. Our students may use the programming framework from *Introduction to Computer Graphics* as their technological basis, but they are free to implement a completely new one.

The remaining two courses target master/graduate students and focus on understanding and implementing state-of-the-art real-time rendering graphics effects and techniques. Compared to each other, these two courses are structured very differently. *Real-Time Rendering* features a relatively open mode, where students are free to choose from a set of sufficiently complex graphics effects, implement them, and put them to good use in a demo application, the technological basis of which students are free to choose as long as they interface with a graphics API directly and not through a 3rd party game engine. *Algorithms for Real-Time Rendering*, on the other hand, tasks students with implementing very specific effects or techniques (or parts thereof) in a framework provided by the course organizers. The code framework that must be used provides a rigid and well-defined track, ruling out the option to use a different technological base than the provided one. The often different approaches that students take for solving a task are then discussed in so-called "solution presentation and group discussion" sessions. While using OpenGL previously, since 2022 this course exclusively uses Vulkan as the underlying graphics API.

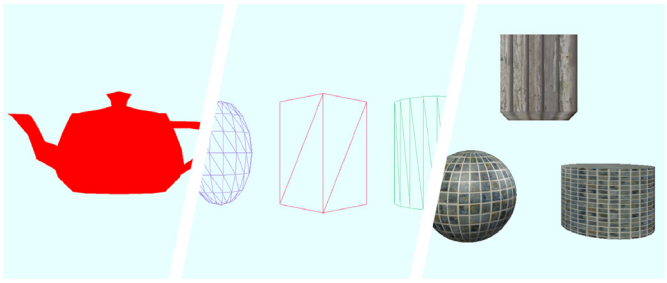


Fig. 2. Screenshots of correct implementations of *Introduction to Computer Graphics* assignments 1, 3, and 5 (from left to right). Task descriptions for each of them are stated in Table 1.

As *Introduction to Visual Computing* does not use graphics APIs and both *Computer Graphics Exercise* and *Real-Time Rendering* allow students to choose their technological basis relatively freely – including the graphics API used to interface with the GPU – we focus mostly on *Introduction to Computer Graphics* and *Algorithms for Real-Time Rendering* in this paper. In Section 4, we describe how we successfully transitioned *Introduction to Computer Graphics* to the Vulkan API, its didactic advantages in Section 5, and the results from a comprehensive student questionnaire about this transition in Section 6. Our experiences using Vulkan in advanced graphics courses are described in Section 7, and student feedback is presented in Section 8.

4. Vulkan in an introductory graphics course

Our course *Introduction to Computer Graphics* targets bachelor/undergraduate students in their third semester and usually constitutes their first encounter with a graphics API. There are five assignments throughout the course, the mandatory tasks of which are listed in Table 1. There are also various bonus tasks for those who want to learn about additional aspects or improve their grade. We provide students with a small OpenGL framework written in C++, which builds and links some helper libraries (GLEW [25], GLFW [26], and GLM [27]), and provides a few utility functions, such as drawing a teapot (with source code hidden, used in the first two assignments), and loading images from file.

If students manage to complete every task on their own, they end up having implemented the essential steps of a basic graphics engine using the OpenGL API: loading (self-generated) geometry data into GPU buffers, providing it to shader programs via vertex attributes, providing all relevant matrices as uniforms to shader programs, and rendering to the screen using custom (self-compiled and linked) shader programs. Should students fail to implement any of these functionalities on their own, we provide them with updated versions of the framework after each assignment's deadline, which contain the functionalities that are required for subsequent tasks. Based on our reference implementation, approximately 1200 LOC have to be written or changed

Table 1
Course assignments and their subtasks. Results of correct implementations of assignments 1, 3, and 5 are shown in Fig. 2.

A#	Tasks
1	Creating a window; setting up a render loop (where a teapot is drawn); reacting to user input
2	Writing, compiling, linking, and using custom shader programs; passing custom transformation matrices as uniforms to shaders; implementing an orbit camera and creating appropriate view matrices for rendering a scene, enabling depth testing
3	Constructing box, cylinder, and sphere geometries as indexed triangle meshes; loading the data into GPU vertex buffer objects and creating vertex array objects; passing vertex positions as vertex attribute to shader programs, enabling primitive culling
4	Adding normals to geometric objects and passing them as additional vertex attributes; implementing Phong illumination [28] combined with Gouraud shading [29] and Phong shading [28] in shaders; illumination from different types of light sources
5	Adding texture coordinates to geometric objects and passing them as additional vertex attributes; loading images into GPU memory, creating mipmaps, sampling from textures in shaders

in C/C++ for the host-side code across all five assignments. In addition, students need to implement approximately 200 LOC in GLSL shaders.

For transitioning the assignments to Vulkan, we wanted to stick to the established OpenGL-based syllabus, even though we anticipated that some tasks would differ quite severely in the implementation requirements. Our goal was to offer students the same tasks, but they would be able to select either OpenGL or Vulkan as their technological basis for the whole course—enabling a smooth transition from one API to the other during this semester and future semesters until we are confident enough to transition permanently. Offering both an OpenGL route and a Vulkan route in the same course with a fixed budget of three ECTS credits [30], we strove to create similar workloads for students of either route. Given a LOC budget of 1200 and the knowledge that drawing a single triangle via the Vulkan API already requires 700 LOC, we had to introduce more utility functions to the framework we provide to students. We tried to find the ideal balance between not sacrificing too much of the learning experience with respect to the Vulkan API and reducing implementation time. In the following, we describe the abstractions that we ended up providing through the Vulkan framework – which we have meanwhile open-sourced as *Vulkan Launchpad [13]* – for each assignment.

In the first assignment, we let students create a Vulkan instance, a surface, select a physical device, create a logical device, queue, and swap chain manually. They directly interface with the Vulkan API for these tasks, because we consider it valuable to let students get in touch with each one of these fundamental types. The remainder of the required initial application setup is abstracted by the framework, namely installing a debug callback, framebuffer, and render pass creation, as well as the creation of the synchronization primitives (semaphores and fences) for swap chain handling [31]. If these had to be set up by students, complex concepts like image layout transitions and synchronization would have to be learned for the first assignment at the beginning of the course already, which we deemed to constitute a too steep learning curve. Students must provide the created handles with additional configuration parameters (e.g., clear color values) to an initialization function. The resulting render loop implementation after completing Assignment 1 leads to C/C++ source code like shown in Listing 1.

Listing 1: Render loop implementation after completing the first assignment. The parameters to the framework initialization function refer to handles of types `VkInstance`, `VkSurfaceKHR`, `VkPhysicalDevice`, `VkDevice`, `VkQueue`, and a custom configuration struct containing required swap chain parameters.

```
// Instance, surface, physical device, logical
// device, queue, and swap chain configuration
// are prepared by students before passing them
// to the framework initialization function:
vklInitFramework(inst, srf, phd, dev, q, swpcfg);

while (!glfwWindowShouldClose(window)) {
 // Handle user input:
 glfwPollEvents();

 // Wait for swap chain img to become available:
 vklWaitForNextSwapchainImage();

 vklStartRecordingCommands();
 vklDrawTeapot();
 vklEndRecordingCommands();

 // Present rendered image to the screen:
 vklPresentCurrentSwapchainImage();
}
```

With this approach, we manage to defer teaching image layout transitions and synchronization to a much later point in the course. Not before Assignment 5, students have to use these for image loading and mipmap creation. The downside is that students do not interface with Vulkan directly in terms of swap chain handling and command buffer recording. Instead, they use framework utility functions (those with "vkl" prefixes). The code of the abstracted functionality in Listing 1 amounts to 300 LOC (not counting the functionality of graphics pipeline creation). Tasking students with implementing these functions on their own during Assignment 1 would have required bigger restructurings of the assignments and most likely would have required the removal of some tasks in later assignments. While it would not be strictly required to draw something to the screen for fulfilling the tasks of Assignment 1, letting the framework draw a red teapot to the current swap chain image provides students with additional feedback on whether their setup code is in a proper state, in addition to possible framework or Vulkan validation error messages.

The creation of custom graphics pipelines is the subject of Assignment 2. The required Vulkan code constitutes another 80 LOC just for graphics pipeline creation, which is why we decided to provide a framework function for it with hard-coded configuration values for many parameters. A few parameters can be configured via a custom struct, which is shown in Listing 2. It only supports vertex and fragment shader stages for the creation of graphics pipelines. Vertex input attribute descriptions translate directly to Vulkan's `VkPipelineVertexInputStateCreateInfo` [24]. It is supposed to be set up for streaming vertex positions during Assignments 2 and 3, to be extended by vertex normals during Assignment 4, and by texture coordinates during Assignment 5. Further configurable parameters are the polygon drawing mode and the primitive culling mode, both relevant for Assignment 3 (Fig. 2 shows the effects of drawing polygon edges as line segments with back-facing triangles being culled). The last member is a set of descriptors stating all resources that are

used in custom vertex or fragment shader programs, which is internally required for pipeline layout creation. For simplicity, we support only one descriptor set, but other than that, we do not abstract or simplify descriptor handling. Instead, students must handle descriptor set layout creation, descriptor set allocation, and descriptor writes manually in Assignments 2 to 5. Several uniform buffers have to be created for storing per-frame uniform data, such as colors and transformation matrices for different objects. Results of correct implementations are shown in Fig. 2.

We refrain from introducing SPIR-V [32], and from letting students compile shader modules on their own, but handle these parts internally in the framework using glslang [33]. This further reduces student workload so that they can focus on shader development. Compilation errors get displayed conveniently in the console. Further functionality that is abstracted by the framework is memory handling for buffers and images. Listing 3 shows the declarations of the relevant framework functions, the implementations of which amount to another 100 LOC. To make students aware of the fact that memory must actually be handled explicitly in Vulkan, we have chosen corresponding expressive function names (including the word "memory") and described them in detail in our documentation.

For enabling depth testing in Assignment 2, a depth buffer image has to be created. Its handle must be provided to the framework's initialization function via the custom swap chain configuration struct, which is mentioned in Listing 1. The declarations of two utility functions for image creation and associated memory handling are shown in Listing 3.

Listing 2: Auxiliary configuration struct with required parameters for custom graphics pipeline creation.

```
struct VklGraphicsPipelineConfig
{
 const char* vertexShaderPath;
 const char* fragmentShaderPath;
 std::vector<VkVertexInputBindingDescription>
    vertexInputBuffers;
 std::vector<VkVertexInputAttributeDescription>
    inputAttributeDescriptions;
 VkPolygonMode polygonDrawMode;
 VkCullModeFlags triangleCullingMode;
 std::vector<VkDescriptorSetLayoutBinding>
    descriptorLayout;
};
```

Listing 3: Convenience functions for creating buffers and images provided by the framework. Associated device memory is handled by the framework internally, opaquely to the user.

```
VkBuffer
vklCreateHostCoherentBufferWithBackingMemory(
    VkDeviceSize, VkBufferUsageFlags);

void
vklCopyDataIntoHostCoherentBuffer(
    VkBuffer, const void*, size_t);

void
vklDestroyHostCoherentBufferAndItsBackingMemory(
    VkBuffer);

VkImage
vklCreateImageWithBackingMemory(
    uint32_t, uint32_t, VkFormat,
    VkImageUsageFlags);

void
vklDestroyImageAndItsBackingMemory(
    VkImage);
```

In Assignment 3, one framework convenience functionality is removed, namely automatic command buffer recording. It is handled opaquely inside the framework for the first two assignments as shown in Listing 1. Starting with Assignment 3, students are required to manually implement command recording in order to pass further vertex attributes to graphics pipelines, and also for transferring image data from buffers into images.

Assignment 4 focuses on shader development and encourages students to use the framework tools they have become acquainted with during the previous assignments. This mostly refers to the creation and proper usage of additional custom graphics pipelines for supporting different illumination methods, and uniform buffers for object data and light-source data.

In Assignment 5, important new concepts are introduced, namely the usage of sampled textures in shaders, synchronization, and image layout transitions. The framework provides functions for loading images from file directly into host-visible buffers. Backing buffers with host-visible memory simplifies their usage since they do not require explicit synchronization, which was exploited in previous assignments. Image memory, however, is allocated in device memory. We explain to students that this leads to more performant rendering, but it also makes synchronization necessary when texture data is transferred from host-visible buffers into the backing memory of images. Students are required to create images, create command buffers, record proper layout transitions via image memory barriers, transfer the image data from a buffer to an image in device memory, submit the command buffers to a queue, and wait for their completion with a fence. All of these operations are to be performed using the Vulkan API directly. Our framework does not provide any further convenience functionality for these tasks of Assignment 5.

5. Didactic advantages of using Vulkan

One side effect of Vulkan's verbosity is that it necessarily reveals more and more underlying hardware details as students progress. Investigative minds are not easily satisfied by following a list of instructions they cannot comprehend. In order for them not to be deterred, Vulkan forces its users to deal with several important concepts discussed in this section that OpenGL does not. Consequently, instructors must address the underlying processes and hardware modules, while in OpenGL, the same use cases may "just work" because the details are handled by drivers internally. Therefore, OpenGL is less likely to encourage investigations of what is going on under the hood. Just by using a low-level graphics API like Vulkan *correctly*, educators are forced to convey more fundamental knowledge about modern GPUs and their architecture.

Vulkan implicitly conveys that switching between different shader programs is never free, as one might come to believe when developing applications based on OpenGL exclusively. Instead, whenever a different shader program shall be used, a whole new graphics pipeline must be created with all its bulk of configuration parameters. The extensive code blocks required to achieve this in Vulkan reflect that changing shaders is rather invasive to the rendering setup and implies potentially heavyweight changes. Users are encouraged by the design of the API to prepare all potentially required pipelines upfront, selecting the appropriate one during render-loop execution. In OpenGL, the driver usually hides this complexity and instead instantiates pipelines dynamically on-demand, reducing the amount of control the user has over the application's runtime performance. For example, when the primitive culling mode is changed, that change can occur during render-loop execution, which might lead to an expensive operation being performed within the render loop.

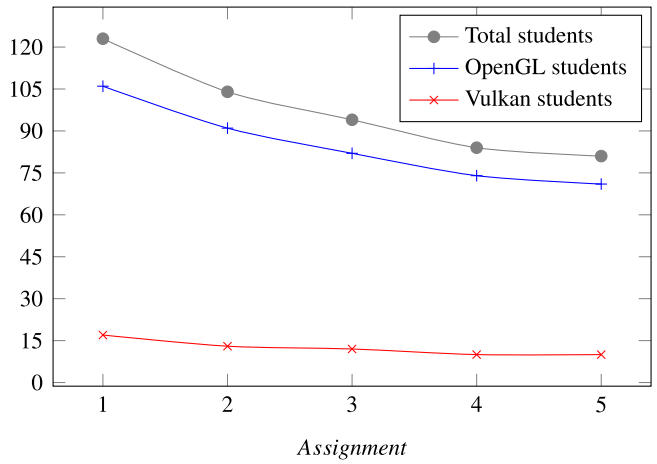
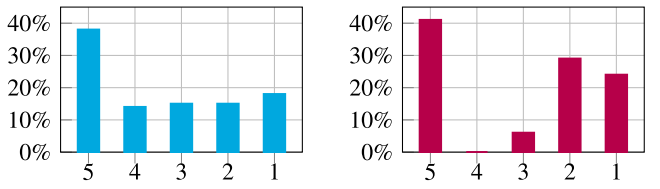


Fig. 3. Number of student submissions for all five assignments.

When a uniform buffer is used to store per-frame data specific to a certain object (e.g., transformation matrices), the same buffer cannot be used for storing per-frame data of another object to be drawn in the same frame in most situations. Recording the drawing of multiple objects into the same command buffer requires the usage of different uniform buffers for the objects' per-frame data since otherwise unwanted effects occur. If, for example, the same host-visible uniform buffer was used for two objects, only the last write to this uniform buffer would succeed due to data being written at queue submission time [24]. These factors force users of the API to think about the reasons why this occurs. Developing these thoughts further, it becomes clear that modern GPUs work in a massively parallel way, which can also mean that both objects from our example are processed in parallel. As such, there must be different uniform buffers – one for each object – accessible during parallel processing of the objects. In OpenGL, again, users do not have to think about these aspects. Uniforms can just be set and used, and rendering "just works", producing the correct result. Users are not forced to think about the modus operandi of modern GPUs and, in the worst case, might think that draw calls are processed in a sequential or host-synchronous manner.

Synchronization, in general, is largely hidden from the API user with OpenGL, bearing the danger of drawing false conclusions about the actual command processing on the hardware. Vulkan, on the other hand, does not hide the responsibility of properly synchronizing commands from its API users, putting the massively parallel nature of modern GPUs into the spotlight. The necessary synchronization must not only be explicitly defined within shaders or between pipeline stages but also between the individual GPU queues that may receive and schedule incoming work. For students who desire to understand and exploit synchronization on a fundamental level, Vulkan provides an additional benefit over older graphics APIs, namely a clearly-defined memory consistency model, which is similar to the well-established C++ memory model [34].

Another area where OpenGL hides vital concepts that affect virtually all modern GPUs is command buffer recording. Commands are simply issued on the fly in OpenGL, completely concealing the possibility of recording and reusing chains of commands, let alone the possible performance implications of command recording. In order to remain efficient, the driver usually caches and organizes these commands, again performing vital work in the user's stead. In Vulkan, all of these concepts are revealed to users so that they are forced to think about the motivation for their presence. From a didactic point of view, achieving a better insight into the inner workings of modern graphics devices can never be wrong.



(a) Grade distribution of OpenGL students (b) Grade distribution of Vulkan students

Fig. 4. Grade distributions of OpenGL students and Vulkan students, where 5 means not passing the course, 4–1 represent positive grades with 1 being the best grade.

6. Student feedback on introductory graphics course

At the beginning of our university winter term in 2021, we offered all of our *Introduction to Computer Graphics* students the choice between OpenGL and Vulkan for implementing the five assignments. This course is mandatory for undergraduate students enrolled in the bachelor program "Media Informatics and Visual Computing" and optional for other students. Since our Vulkan framework was brand-new and we did not have any prior student feedback, we deployed a corresponding warning message and told them that they should expect up to 150% of required effort compared to the OpenGL route. From a total of 123 initial students, 17 (14%) opted for the Vulkan route. 81 students completed the course, among them 10 students (12%) who chose the Vulkan route. In this section, we present insights from a detailed questionnaire that was completed by 52 OpenGL students (73% of total OpenGL students who completed the course) and by 9 Vulkan students (90% of total Vulkan students who completed the course). A diagram showing the development of numbers of student submissions over the whole course is shown in Fig. 3.

Given our warning about the potentially increased effort using Vulkan combined with reduced effort having been the top motivator for the students who chose OpenGL (see Table 3) suggests that students who chose Vulkan may generally have been more strongly motivated for doing the course. Based on the grade distributions shown in Fig. 4, this does not necessarily mean that Vulkan students were more skilled in general, since the relative amount of negative grades is almost the same for both groups of students. The positive grades were shifted more towards the better grades in the Vulkan group, which can be attributed to higher motivation levels. Students of both groups made use of consultation hours approximately equally throughout the course.

Fig. 5 shows how students who chose the OpenGL route compare to the students who chose the Vulkan route in terms of how they perceived the five assignments according to the categories *workload*, *difficulty*, and *usefulness* of the respective API. The wording of our questions was as follows:

- "How was the workload of Assignment X in your opinion?" Answers range from "Almost nothing to do" (−2), over "Adequate workload" (0), to "Too much effort" (2).
- "How was the difficulty of Assignment X?" Answers range from "Too easy" (−2), over "Difficulty was ideal" (0), to "Too hard" (2).
- "Do you think that the skills you picked up during Assignment X will be useful in your future career?" Answers range from "Not useful at all" (−2) to "Extremely useful" (2).

Interestingly, the assessments of both groups of students are similar for Assignments 2 to 5, which is a good sign since it indicates that the transition to Vulkan did not have a major

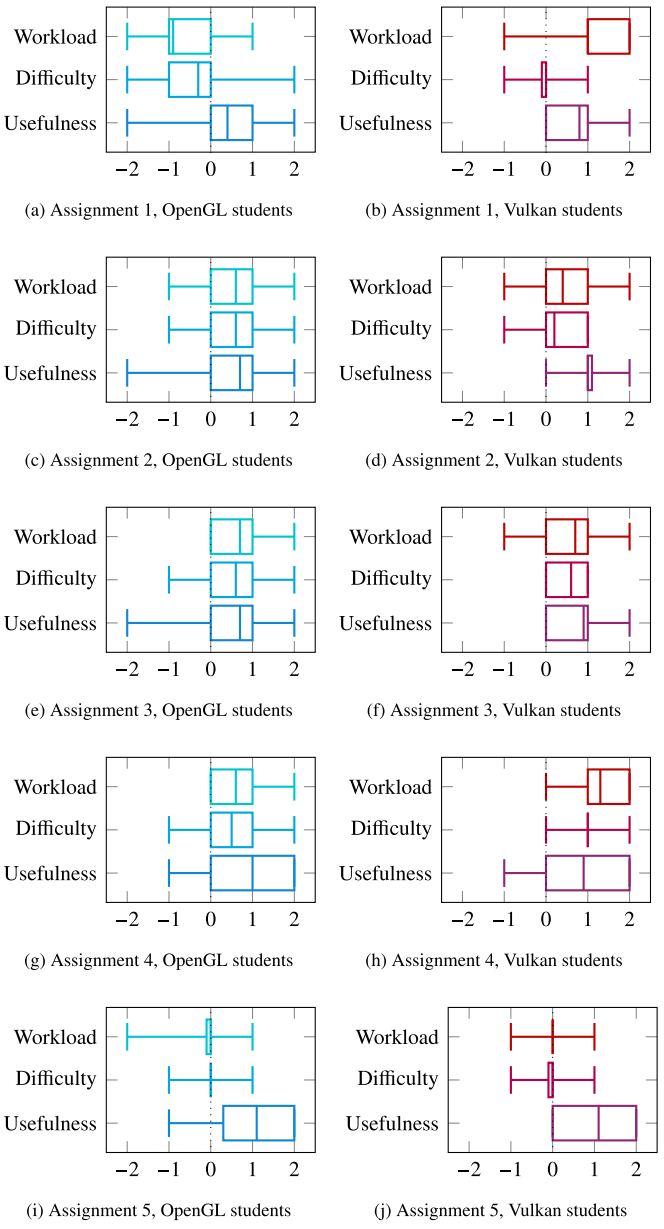
Table 2
Reasons of Vulkan students for choosing Vulkan. The second column states the percentage of Vulkan students who declared the respective reason.

Reasons given by students for choosing Vulkan	%
Wanted to learn this API	100%
Provided framework seemed to be in a better state	33%
Vulkan is more relevant for game development	11%
More modern API	11%

Table 3
Reasons given by students for choosing OpenGL. The second column states the percentage of OpenGL students who declared the respective reason.

Reasons for choosing OpenGL	%
Wanted to reduce the effort required to do this course	60%
Wanted to learn this API	38%
Provided framework seemed to be in a better state	25%
The task descriptions seemed to be clearer	25%
Already had experience with this API	21%

impact in these regards. Only for Assignment 1 we can observe a higher rating of workload and difficulty for the Vulkan version, which is unfortunate, as it might have contributed to the higher dropout rate of Vulkan students (24%) between Assignments 1 and 2 compared to the dropout rate of OpenGL students (14%). For the subsequent assignments, dropout rates were similar (see Fig. 3) for both groups. The box plots in Fig. 5 represent students' perceived workload, i.e., whether they felt that the necessary workload was appropriate for an assignment or not. The box plots in Fig. 6 present students' estimates (or actual amounts) of hours spent on the mandatory tasks per assignment. It can be inferred that the initial effort for the Vulkan version of Assignment 1 was higher than for its OpenGL counterpart. The time investments for all other assignments, though, were similar for both groups. The minimum outliers point towards a slightly higher baseline in terms of required effort across all assignments for the Vulkan assignments. Interestingly, Vulkan students required less time for Assignments 2 and 3 than OpenGL students. Across all assignments, the maximum outliers indicate a lower upper bound of effort. This could be an effect of possibly higher motivation levels among Vulkan students. On the other hand, 0% of students had prior experience with Vulkan, while 21% of OpenGL students already had prior experience with the API of their choice. Tables 2 and 3 list further reasons for students deciding in favor of an API. With 60%, the strongest motivator of OpenGL students was to avoid the potentially higher workload of the Vulkan route. Only 38% of OpenGL students deliberately wanted to learn the OpenGL API, so there seems to be a lot of potential for getting students interested in learning a different API. Our newly introduced Vulkan framework seems to have deterred 25% of OpenGL students, but on the other hand, 33% of Vulkan students assessed it to be in a better state than its battle-tested OpenGL counterpart. Students had the chance to take a look at both frameworks and assignment descriptions of both routes before deciding upon which route to take. Although we provided much more extensive and detailed task descriptions for the Vulkan assignments, 25% of OpenGL students declare them as a reason for their choice in favor of OpenGL. None of the Vulkan students mentioned the task descriptions as a deciding factor, but 78% of Vulkan students found them helpful even as a learning resource for Vulkan overall, as shown in Table 4. The same amount of students in this group found the Vulkan specification helpful, while *vulkan-tutorial.com* was mentioned as a helpful learning resource by the largest number of students. With the intent of explaining fundamental Vulkan concepts in a vivid and comprehensible way, we started producing Vulkan lectures and provided them to our students throughout the course via the "Vulkan Lecture Series" [35]. Unfortunately,



(a) Assignment 1, OpenGL students (b) Assignment 1, Vulkan students
(c) Assignment 2, OpenGL students (d) Assignment 2, Vulkan students
(e) Assignment 3, OpenGL students (f) Assignment 3, Vulkan students
(g) Assignment 4, OpenGL students (h) Assignment 4, Vulkan students
(i) Assignment 5, OpenGL students (j) Assignment 5, Vulkan students

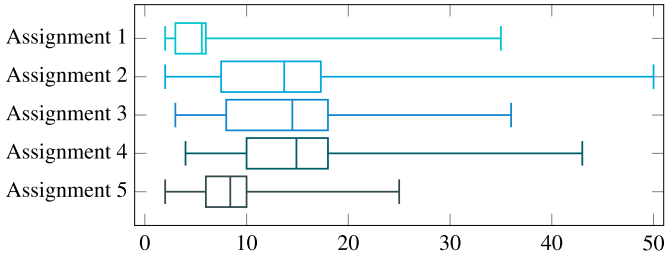
Fig. 5. Students' assessments of the assignments, with respect to workload, difficulty, and usefulness; all rated on a scale from low (−2) to high (2), where 0 means adequate, perfect, and medium, respectively for the different categories.

Table 4
Reported learning resources of Vulkan students.

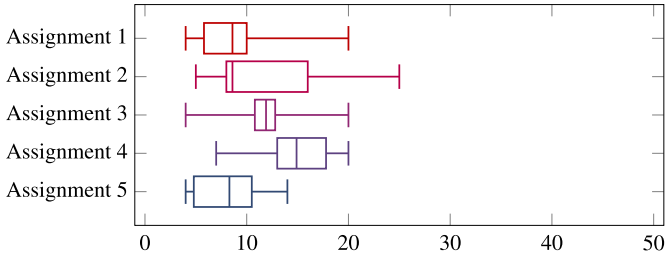
Vulkan resource	%
vulkan-tutorial.com [11]	89%
Vulkan specification [24]	78%
Assignment description documents	78%
"Vulkan Lecture Series" on YouTube [35]	44%
Official Khronos examples [31]	33%
Sascha Willems' tutorials and examples [36,37]	22%

some episodes were running late and, thus, were not available to our students timely. We hope that providing these lectures right from the beginning of the semester will lead to better and faster learning success in the future.

As far as problems during implementation of the assignments are concerned, a larger amount of OpenGL students mentioned problems with graphics API usage than their colleagues on the

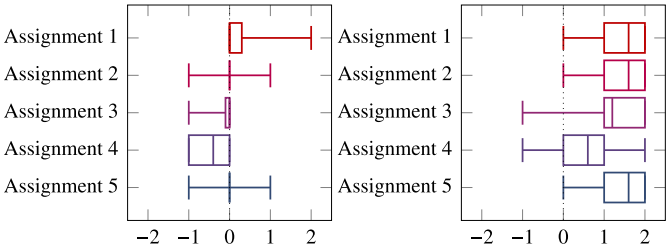


(a) Hours required per OpenGL assignment



(b) Hours required per Vulkan assignment

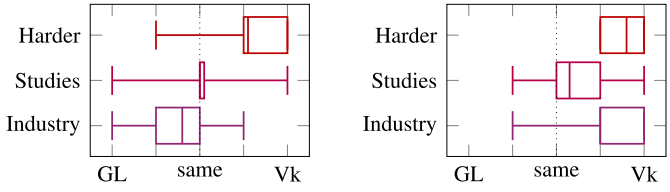
Fig. 6. Hours it takes to complete all mandatory tasks per assignment, as reported by students.



(a) Level of abstraction (b) Learning experience

Fig. 7. Vulkan students' assessments of the assignments, with respect to the framework's level of abstraction, and their API learning experience. The level of abstraction is rated from a too low level of abstraction (−2), over a perfect balance between learning the API and saving time (0), to a too high level of abstraction (2). Learning experience ranges from having learned nothing (−2), over a medium amount (0), to having learned a lot (2) about the Vulkan API.

Vulkan route. Graphics API usage turned out to be problematic to 52% of all OpenGL students (see Table 5), while only 22% of Vulkan students reported problems with direct Vulkan API usage (see Table 6). One reason for the high percentage of students declaring problems with C/C++ programming stems from the fact that many students get in touch with C++ programming for the first time during this course in their bachelor program. Many had mainly experience with the Java programming language and had not used C or C++ before. Although the Vulkan framework contains a lot more functionality than its OpenGL counterpart, and Vulkan students must interface with the provided framework on more occasions, framework usage was not declared as being problematic by a larger fraction of Vulkan students when compared to the fraction of OpenGL students reporting problems with framework usage (see Tables 5 and 6). Overall, Vulkan students stated to be pretty happy with the framework's level of abstraction (see Fig. 7(a)). Some had even hoped for a lower level of abstraction for Assignment 1, although the workload of Assignment 1 was rated as being too high (see Fig. 5(b)). These students were eager to acquire the knowledge about the details of the abstracted functionality. Nevertheless, Vulkan students were generally satisfied with their learning experience (see Fig. 7(b)).



(a) OpenGL students (b) Vulkan students

Fig. 8. "Harder" refers to the question: "Which API do you think is harder to learn, OpenGL or Vulkan?". "Studies" refers to the question: "Which API do you think will be more useful during your studies, OpenGL or Vulkan?". "Industry" refers to the question: "Which API do you think would be more useful for working in the industry, OpenGL or Vulkan?".

Table 5
Biggest problems of OpenGL students during development.

Problems with the OpenGL route	%
Graphics API usage (direct OpenGL API usage)	52%
Programming in C/C++	42%
Using the provided OpenGL framework	35%

Table 6
Biggest problems of Vulkan students during development.

Problems with the Vulkan route	%
Programming in C/C++	56%
Using the provided Vulkan framework	33%
Graphics API usage (direct Vulkan API usage)	22%

Fig. 8 shows that both groups of students think that Vulkan is much harder to learn than OpenGL. OpenGL students are indecisive whether OpenGL or Vulkan might be more helpful in their further studies, while Vulkan students lean towards Vulkan in that regard. Each group of students thinks that their respective API of choice will be more useful for working in the industry, while Vulkan students show a higher degree of confidence. A multi-platform framework for Windows and Linux was strongly requested by some of our students.

This concludes our discussion of student feedback that we obtained during *Introduction to Computer Graphics* in 2021. In the next sections, we describe Vulkan utilization in our more advanced graphics courses, which frameworks we use, and we present further student feedback on these advanced courses.

7. Vulkan in advanced graphics courses

The mode that we employ in our advanced graphics course *Algorithms for Real-Time Rendering*, targeted to graduate students, tasks students with the implementation of a set of well-defined and well-bounded state-of-the-art real-time rendering techniques and effects. Topics include proper implementation of normal mapping using tangent space, handling a large number of light sources, hardware tessellation, view-frustum and backface culling in tessellation control shaders, dynamically adaptive levels of tessellation, deferred shading, deferred shading in combination with multisample anti-aliasing, manual multisample resolve in compute shaders, tile-based deferred shading, physically based shading, screen-space ambient occlusion, tone mapping, temporal anti-aliasing, and real-time ray tracing using ray query and ray tracing pipelines. While using OpenGL previously for all tasks (except ray tracing), then starting the transition to Vulkan for some tasks while leaving some in OpenGL, we switched to Vulkan completely for all tasks in 2022. This allowed us to address some advanced low-level GPU topics in teaching that were impossible to cover with the OpenGL API: Real-time ray tracing is

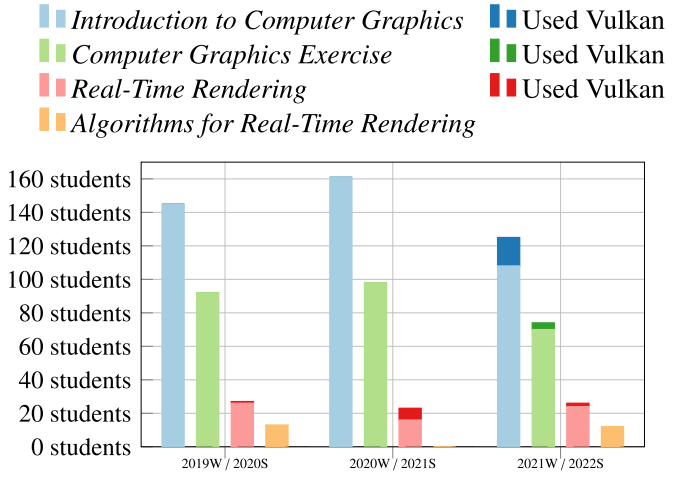


Fig. 9. Student numbers in the different courses across three years, grouped by winter term and subsequent summer term: 2019W/2020S, 2020W/2021S, and 2021W/2022S.

not supported at all by the OpenGL API, but it is with Vulkan. On a more fundamental level, fine-grained synchronization and its effects can only be analyzed and discussed properly with a low-level API like Vulkan since all the synchronization primitives which would allow fine-grained synchronization are missing from OpenGL. Having the possibility to discuss them enables us to teach low-level concepts which we consider being very important for acquiring in-depth knowledge about GPU programming. A further benefit of using the Vulkan API is the explicit and precise specification of when multisample resolve operations are performed by the GPU along with appropriate framebuffer attachment usage and synchronization. While inclusion of more low-level concepts like buffer sub-allocation would have been interesting topics to cover additionally, we were unable to fit more such topics into the scope of this course—not least because its main focus is on rendering algorithms.

Vulkan's high verbosity poses two main challenges: First, it requires a lot of code to be implemented and second, it can constitute quite a big challenge to present such large amounts of code to other people. To counteract both of these challenges, we do not use the Vulkan API directly, but instead we use two frameworks that were developed by our research group and help to solve these problems pretty well according to the feedback that we received from our students: *Auto-Vk [14]* and *Auto-Vk-Toolkit [15]*. The former provides a low-level abstraction layer over the Vulkan API (more precisely over Vulkan-Hpp [38]), enabling code that is much more concise and possibly faster to write. To accomplish that, it uses many advanced C++ features and some sensible defaults, which can be overwritten by users. *Auto-Vk-Toolkit* adds a multitude of useful functionality to it like window system integration, input handling, render-loop handling, model and image loading, and utilities like asset management, serialization, and shader hot-reloading. These functionalities help to reduce development and project setup efforts, letting users focus on Vulkan development.

During our grading events – called "solution presentation and group discussion" events – we ask students to present their solutions to the group of students on a big screen. The big benefit of using *Auto-Vk* manifested in the effect that in many cases, the relevant code for a task could be shown on one single screen. For example, a graphics pipeline can be created like shown in Listing 4 with just a few LOC, while raw Vulkan code for the same task can easily require 80 LOC or more.

Listing 4: C++ source code for creation of a graphics pipeline using *Auto-Vk*. The parameters refer to shader files, specify vertex buffer input binding locations matched with a data format and shader input location, a renderpass, front-facing configuration, and several descriptor bindings to various resources.

```
using namespace avk; using namespace glm;

// Create a new graphics pipeline consisting of
// a vertex shader and a fragment shader:
auto p = context().create_graphics_pipeline_for(
 // Specify shaders for this pipeline:
 vertex_shader("my_shader.vert"),
 fragment_shader("my_shader.frag"),

 // Specify buffer bindings to target locations:
 from_buffer_binding(0)
   ->stream_per_vertex<vec3>()->to_location(0),
 from_buffer_binding(1)
   ->stream_per_vertex<vec2>()->to_location(1),

 // Use a renderpass created previously:
 renderpass,
 // Further config parameters:
 cfg::front_face::
    define_front_faces_to_be_counter_clockwise(),

 // Define resource bindings:
 descriptor_binding(0, 0, mMaterials),
 descriptor_binding(1, 0, mUniformsBuffer),
 descriptor_binding(1, 1, mLightsBuffer)
);
```

Listing 5: C++ source code using *Auto-Vk* for declaring that a framebuffer attachment in a certain format shall be cleared on load, used as depth/stencil attachment in the first subpass, used as input attachment bound to location 1 in the second subpass, used as depth/stencil attachment in the third subpass and resolved to the attachment at index 3 after the third subpass.

```
using namespace avk;

attachment::declare(
  vk::Format::eD32Sfloat,
  on_load::clear,
  usage::depth_stencil
    >> usage::input(1)
    >> usage::depth_stencil+usage::resolve_to(3),
  on_store::dont_care)
```

Listing 6: C++ source code using *Auto-Vk* for creating a global memory barrier that synchronizes a transfer operation (making its write accesses available) with a compute pipeline (ensuring the memory is visible for read access).

```
using namespace avk;

auto barrier = sync::global_memory_barrier(
  stage::transfer >> stage::compute_shader,
  access::transfer_write >> access::shader_read
)
```

Listing 4 shows that *Auto-Vk* manages to require fewer LOC by establishing some default configuration but still enables further configuration options, which can be added to the function call through C++ variadic templates. Another example is attachment declaration for renderpass creation as shown in Listing 5. It allows expressively specifying the attachment usage across different subpasses and also where a resolve operation should happen precisely. The setup code is arguably as concise as it can be for that purpose while not hiding any conceptual low-level details. A final example presented in this paper is the code

Table 7
Students' answers to the question whether they liked that some/all tasks of *Algorithms for Real-Time Rendering* were based on the Vulkan API.

Did you like tasks based on Vulkan?	2020	2022
Preferred not to use Vulkan.	12.5%	0%
Don't care.	12.5%	0%
Using Vulkan was good.	25%	71%
Using Vulkan was great.	50%	29%

Table 8
Students' answers about their Vulkan experience prior to starting the course *Algorithms for Real-Time Rendering*.

Prior experience with Vulkan	2020	2022
Zero.	50%	57%
A little bit.	12.5%	14%
Used it for a small project.	25%	29%
Used it for a bigger project	12.5%	0%
A lot/used it for several projects.	0%	0%

Table 9
Students' answers whether they had the feeling that *Auto-Vk-Toolkit* helped to make Vulkan development easier.

Is *Auto-Vk-Toolkit* helpful w.r.t. Vulkan?	2020	2022
No, makes it harder.	0%	0%
Barely helps.	12.5%	0%
Helps a bit.	25%	57%
Helps a lot.	62.5%	43%

for establishing a global memory barrier in a concise manner as shown in Listing 6. Also in this case, the corresponding raw Vulkan code would require many more LOC.

Besides being well suited for teaching purposes, the two frameworks are furthermore intended to be used for rapid prototyping and general Vulkan development. We have successfully used them in our published research, like for our work on exploiting conservative meshlet bounds using graphics mesh pipelines [3]. Some of their advantageous properties are reflected in student feedback on our advanced graphics courses.

8. Student feedback on advanced graphics courses

In this section, we present results from student feedback based on anonymous questionnaires about their individual assessments and experiences with Vulkan in advanced graphics courses. Fig. 9 shows the average student numbers in the last three years. It can be seen that the vast majority of our students still used the OpenGL API. They can choose between OpenGL and Vulkan in *Introduction to Computer Graphics*, as described in Section 6, since winter term 2021 (2021W). The effects on the subsequent courses *Computer Graphics Exercise* and *Real-Time Rendering*, where students are free to choose any graphics API, are not yet measurable. *Algorithms for Real-Time Rendering* – which is conducted bi-yearly – dictates an API to be used for each assignment. In 2020, 50% of the assignments were based on OpenGL and 50% were based on Vulkan. In 2022, 100% of the assignments were based on Vulkan.

In our more advanced courses, we noticed a rise in interest in the Vulkan API in recent years. In 2020, 60% of students answered that they would have liked to learn more about Vulkan in the context of *Real-Time Rendering*. This number increased to 89% in 2021, but these numbers might not reflect all students too well since only 22% and 35% of participating students provided feedback, respectively. Interestingly, in 2020, 29% of students chose the Vulkan API for implementing their demos. Of this group, 83% used *Auto-Vk-Toolkit* to interface with the Vulkan API. In 2021, the number of students using Vulkan dropped back to 7%, which is similar to previous years (4% in 2019).

Basing 100% of the assignments of *Algorithms for Real-Time Rendering* on Vulkan in 2022 led to a generally increased engagement with the Vulkan API. Additionally, we also slightly increased the number of Vulkan-centric tasks, so that students got to learn more aspects of the API. In 2020, 62% of students provided feedback through a questionnaire. In 2022, 58% provided feedback. Almost all students appreciated that our tasks were based on Vulkan as Table 7 shows. This is quite remarkable given the fact that more than half of participating students had absolutely no prior Vulkan experience, and none rated themselves as having a lot of prior Vulkan experience. The exact percentages are listed in Table 8.

Students replied that *Auto-Vk-Toolkit* eases development with the Vulkan API, as attested by the numbers in Table 9. We believe that the shift from mostly "Helps a lot" in 2020 towards "Helps a bit" in 2022 does not indicate a decline in framework quality but is merely a side-effect of us putting more focus on low-level details in certain new or reworked Vulkan-centric tasks in 2022. For example, students had to pay attention to establishing proper synchronization between commands or renderpass subpasses. Some students would have preferred a higher level of API abstraction in *Auto-Vk*, while others would have preferred less abstraction to learn direct Vulkan API usage. Most students seem to agree, though, that using Vulkan directly would have imposed a much too high workload in the context of this course. One student pointed out that *Auto-Vk* is especially well suited for teaching purposes and that the code presentations and discussions during our "solution presentation and group discussion" events would probably have been cumbersome without it.

9. Conclusion

We successfully employed Vulkan for teaching the use of a real-time graphics API in an introductory course and for teaching selected state-of-the-art techniques and low-level GPU concepts in advanced courses. Abstracting some functionality of early assignments was key to enabling a manageable and fair workload. Flattening the learning curve of Vulkan for first-time graphics API users enabled us to provide a similar challenge as previously established OpenGL assignments. But also in advanced courses, the use of a framework that abstracts the Vulkan API and reduces implementation effort turned out to be crucial to stay within sensible boundaries in terms of student workload. However, introductory and advanced courses have different requirements in terms of Vulkan API abstraction. Therefore we propose to use *Vulkan Launchpad [13]* for the former to hide several complex concepts for Vulkan first-time users—and for the latter we propose using *Auto-Vk [14]* and *Auto-Vk-Toolkit [15]*, which do not hide any fundamental Vulkan concepts but just aim to make development more efficient and code more concise.

Interestingly, the biggest hurdle for many students in introductory graphics courses was C/C++ usage, constituting a bigger problem for Vulkan students since they had to write more code in their first assignment. More efficient C/C++ learning resources and lectures should allow students to focus better on graphics API usage. The biggest hurdles for students with respect to graphics API usage in our advanced courses seemed to be some of the advanced low-level concepts like synchronization. Since the vast majority (presumably all) of students in advanced courses did not learn graphics programming using the Vulkan API, but instead using a higher-level API like OpenGL, many of them had to learn concepts like fine-grained synchronization at a later point in time in the context of our courses. We think that teaching Vulkan from the start will have a positive effect on our students for becoming proficient users of modern graphics APIs and, thereby,

in more advanced courses when they encounter Vulkan again. Using a low-level API enables students to learn about the massively parallel operation mode of modern GPUs early in their visual computing education. Our evaluation has shown that students appreciate the skills and knowledge they picked up through using the Vulkan API. We believe that teaching Vulkan is both viable and beneficial to students who aim to become competent practitioners of visual computing. While the transition may be challenging, it appears to be a worthwhile investment to provide students with future-proof education.

CRediT authorship contribution statement

Johannes Unterguggenberger: Software, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization, Project administration. **Bernhard Kerbl:** Conceptualization, Writing – review & editing. **Michael Wimmer:** Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The original OpenGL framework for Introduction to Computer Graphics was created by Lukas Gersthofer and Bernhard Steiner. We could build upon their great work when creating the Vulkan counterpart. Lukas Gersthofer provided valuable feedback and helped to improve the Vulkan framework's code through code reviews. Lukas Geyer and Alexander Cech provided invaluable help with Algorithms for Real-Time Rendering, also helping to shape Auto-Vk and Auto-Vk-Toolkit. Hamed Jafari, Martin Rumpelnik, Lukas Herzberger, Jakob Pernsteiner, Wolfgang Rumpler, Stefan Fiedler, Andreas Wiesinger, and Simon Fraiss helped to improve Auto-Vk and Auto-Vk-Toolkit through their contributions. We thank Anna Sebernegg and Wolfgang Rumpler for providing a screenshot of their game "Hyper Race" for usage in Fig. 1. We thank Hiroyuki Sakai for providing feedback and TikZ expertise. This work was supported by the Austrian Science Fund (FWF), project no. F77.

References

[1] Unterguggenberger J, Kerbl B, Steinberger M, Schmalstieg D, Wimmer M. Fast multi-view rendering for real-time applications. In: Frey S, Huang J, Sadlo F, editors. Eurographics symposium on parallel graphics and visualization. Eurographics; 2020, p. 13–23. http://dx.doi.org/10.2312/pgv.20201071, URL: https://www.cg.tuwien.ac.at/research/publications/2020/unterguggenberger-2020-fmvr/.

[2] The Khronos® Group Inc. Vulkan — cross platform 3D graphics. 2022, https://www.vulkan.org [Accessed 21-Jan-2022].

[3] Unterguggenberger J, Kerbl B, Pernsteiner J, Wimmer M. Conservative meshlet bounds for robust culling of skinned meshes. Comput Graph Forum 2021;40(7):57–69. http://dx.doi.org/10.1111/cgf.14401, URL: https://www.cg.tuwien.ac.at/research/publications/2021/unterguggenberger-2021-msh/.

[4] Microsoft Corporation. Directx graphics and gaming. 2022, https://docs.microsoft.com/en-us/windows/win32/directx [Accessed 21-Mar-2022].

[5] Apple Inc. Metal. Accelerating graphics and much more. 2022, https://developer.apple.com/metal [Accessed 21-Mar-2022].

[6] The Brenwill Workshop Ltd. KhronosGroup/moltenvk. 2022, https://github.com/KhronosGroup/MoltenVK [Acc. 21-Mar-2022].

[7] The Khronos® Group Inc. Khronos members - the khronos group inc. 2022, https://www.khronos.org/members/list [Accessed 21-Mar-2022].

[8] The Khronos® Group Inc. VK_NV_ray_tracing(3) manual page. 2018, https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_NV_ray_tracing.html [Accessed 21-Mar-2022].

[9] The Khronos® Group Inc. VK_KHR_ray_tracing_pipeline(3) manual page. 2020, https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_KHR_ray_tracing_pipeline.html [Accessed 21-Mar-2022].

[10] de Vries J. LearnOpenGL - hello triangle. 2022, https://learnopengl.com/Getting-started/Hello-Triangle [Accessed 22-Jan-2022].

[11] Overvoorde A. Vulkan tutorial. 2022, https://vulkan-tutorial.com [Accessed 22-January-2022].

[12] Unterguggenberger J, Kerbl B, Wimmer M. The road to vulkan: Teaching modern low-level APIs in introductory graphics courses. In: Eurographics 2022 - education papers. 2022, p. 31–9, URL: https://www.cg.tuwien.ac.at/research/publications/2022/unterguggenberger-2022-vulkan/.

[13] Research Unit of Computer Graphics | TU Wien. Vulkan launchpad. 2022, https://github.com/cg-tuwien/VulkanLaunchpad.

[14] Research Unit of Computer Graphics | TU Wien. Auto-vk. 2022, https://github.com/cg-tuwien/Auto-Vk.

[15] Research Unit of Computer Graphics | TU Wien. Auto-vk-toolkit. 2022, https://github.com/cg-tuwien/Auto-Vk-Toolkit.

[16] Suselo T, Wünsche BC, Luxton-Reilly A. The journey to improve teaching computer graphics: A systematic review. In: Proc. 25th int. conf. comput. educ.. 2017, p. 361–6.

[17] Chen M, Xu Z, Rippin W. On the pedagogy of teaching introductory computer graphics without rendering API. Eurographics technical report series EG 2018, The Eurographics Association; 2018, p. 47–50.

[18] Fink H, Weber T, Wimmer M. Teaching a modern graphics pipeline using a shader-based software renderer. Comput Graph 2012;37(1–2):12–20.

[19] Balreira DG, Walter M, Fellner DW. What we are teaching in introduction to computer graphics. In: Eurographics (education papers). 2017, p. 1–7.

[20] Reina G, Müller T, Ertl T. Incorporating modern OpenGL into computer graphics education. IEEE Comput Graph Appl 2014;34(4):16–21.

[21] Shreiner D, Sellers G, Kessenich JM, Licea-Kane BM. OpenGL programming guide: The official guide to learning OpenGL, version 4.3. 8th ed.. Addison-Wesley Professional; 2013.

[22] Sellers G, Wright RS, Haemel N. OpenGL superbible: Comprehensive tutorial and reference. 7th ed.. Addison-Wesley Professional; 2015.

[23] AMD. V-EZ. 2018, https://github.com/GPUOpen-LibrariesAndSDKs/V-EZ [Accessed 22-Oct-2022].

[24] The Khronos Group Inc. Vulkan 1.2.203 - a specification (with all registered vulkan extensions). 2021, www.khronos.org/registry/vulkan/specs/1.2-extensions/html [Accessed 24-Jan-2022].

[25] Stewart N. GLEW - the OpenGL extension wrangler library. 2022, https://github.com/nigels-com/glew [Accessed 16-Mar-2022].

[26] Löwy C. An OpenGL library | GLFW. 2021, https://www.glfw.org/ [Accessed 22-Jan-2022].

[27] G-Truc Creation. OpenGL mathematics (GLM). 2021, https://github.com/g-truc/glm [Accessed 22-Jan-2022].

[28] Phong BT. Illumination for computer generated pictures. Commun ACM 1975;18(6):311–7.

[29] Gouraud H. Continuous shading of curved surfaces. IEEE Trans Comput 1971;100(6):623–9.

[30] European Union. European credit transfer and accumulation system (ECTS). 2022, https://education.ec.europa.eu/levels/higher-education/inclusion-connectivity/european-credit-transfer-accumulation-system [Accessed 21-Mar-2022].

[31] The Khronos® Group Inc. KhronosGroup/Vulkan-Samples: one stop solution for all vulkan samples. 2022, https://github.com/KhronosGroup/Vulkan-Samples [Accessed 24-Jan-2022].

[32] The Khronos® Group Inc. SPIR overview - the khronos group inc. 2022, https://www.khronos.org/spir [Accessed 24-Jan-2022].

[33] The Khronos® Group Inc. KhronosGroup/glslang: khronos-reference front end for GLSL/ESSL, partial front end for HLSL, and a SPIR-v generator. 2022, https://github.com/KhronosGroup/glslang [Accessed 24-Jan-2022].

[34] Hector T. Comparing the vulkan SPIR-V memory model to C++'s. 2018, https://www.khronos.org/blog/comparing-the-vulkan-spir-v-memory-model-to-cs [Accessed 21-Mar-2022].

[35] TU Wien, Institute of Visual Computing and Human-Centered Technology. Vulkan lecture series - computer graphics at TU wien. 2021, https://www.youtube.com/watch?v=tLwbj9qys18&list=PLmIqTlJ6KsE1Jx5HV4sd2jOe3V1KMHHgn.

[36] Willems S. Vulkan tutorial. 2022, https://www.saschawillems.de/tags/vulkan-tutorial/ [Accessed 16-Mar-2022].

[37] Willems S. SaschaWillems/Vulkan: examples and demos for the new vulkan API. 2022, https://github.com/SaschaWillems/Vulkan [Accessed 16-Mar-2022].

[38] The Khronos® Group Inc. KhronosGroup/Vulkan-Hpp, open-source vulkan C++ API. 2022, https://github.com/KhronosGroup/Vulkan-Hpp [Accessed 18-Sep-2022].