The Road to Vulkan: Teaching Modern Low-Level APIs in Introductory Graphics Courses

Johannes Unterguggenberger, Bernhard Kerbl and Michael Wimmer

TU Wien

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. Various properties of this API and overall trends suggest that Vulkan could soon replace OpenGL in many areas. 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 an introductory graphics course side-by-side with conventional OpenGL. A final survey enables us to draw conclusions about perceived workload, difficulty, and students' acceptance of either approach and identify suitable conditions and recommendations for teaching Vulkan to undergraduate students.

CCS Concepts

• ***Social and professional topics*** → ***Student assessment;*** • ***Computing methodologies*** → *Rasterization;*

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 [UKS*20], 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 [Khr22vk]. At that time, we decided to stick with OpenGL, primarily due to its far-reaching support in GPU profilers, quick prototyping ability, and less verbose written code. For our work on computing and exploiting conservative meshlet bounds [UKPW21], we switched to Vulkan, since it abstracts the hardware on a lower level than OpenGL, offering more control over the hardware, which is particularly desirable in the field of real-time rendering. With rising proficiency in this new API and its continuously improved tooling, we eventually noticed an increase in productivity thanks to its invaluable validation layers and potent debugging features. Vulkan provided us with more insights and more 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 at our university. 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 two major modern, low-level graphics APIs: DirectX 12 [Msf22] and Metal [Apl22]. While all three of these 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 [Bre22] 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 [Khr22me]. They all contribute to

DOI: 10.2312/eged.20221043

shape and maintain the Vulkan API, while DirectX and Metal are proprietary standards, defined and controlled by one 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. E.g., hardware-accelerated ray tracing was available through an NVIDIA-specific extension [Khr18ray] only one month after its availability in DirectX and has later been standardized [Khr20ray]. Given these conditions, Vulkan is the only sensible choice in university 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 [dVri22]. 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 [Ove22]. 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 undergraduate 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—where conveying fundamental knowledge constitutes a prime goal of higher education, anyways.

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 a whole new course (which would have implied a whole series of changes to subsequent courses in the curriculum), sticking to OpenGL in bachelor programs and switching to Vulkan in master programs, or trying to introduce Vulkan as early as possible as a complete replacement of OpenGL. We ultimately decided to go with the latter strategy, as it turned out to be minimally invasive curriculum-wise. Furthermore, we argued that it would constitute the highest benefit for undergraduate students, *if* we would succeed in introducing Khronos' new low-level API properly. That latter condition remained the big unknown since we expected learning and teaching workloads to rise and hence required a fail-safe. Consequently, in 2021, we doubled our teaching efforts and offered students the choice to stick with battle-tested OpenGL or embrace the new, exciting, uncertain, and potentially effortful Vulkan route.

Having finalized the conduction of the course in question, we can conclude that the Vulkan route was much less bumpy than we initially expected, and therefore, we want to describe a pragmatic route for transitioning teaching to Vulkan for undergraduate students who possess fundamental knowledge of linear algebra, programming, a basic understanding of rasterization and aim to use a graphics API for the first time for real-time rendering. In this paper, we describe the changes that we have made to our "Introduction to Computer Graphics" course, outline our own experiences, and present results from a student questionnaire, providing detailed insights into our transition to Vulkan from the students' perspectives.

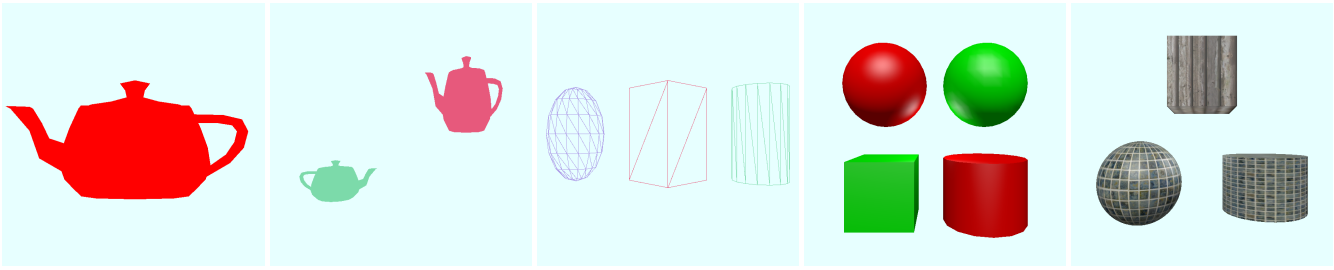
2. Related Work

Fundamental difficulties of students learning computer graphics and potential countermeasures are described by Suselo et al. [SWL17]. 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. [CXR18], 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 has been described by Fink et al. [FWW12]. 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. [BWF17], it can be concluded that OpenGL was the most widely used graphics API in university education, given the absence of any mention of other graphics APIs in 2017. 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 already have migrated. Experiences with the transition from legacy OpenGL to modern OpenGL in university education are described by Reina et al. [RME14]. 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" [SSKL13] or the "OpenGL Superbible" [SWH15] are not yet available for Vulkan. Early attempts to provide additional abstraction or simplify the interface had only limited success[AMD18]. However, Vulkan as an API is still evolving. Recent additions to the SDK, such as the `VK_KHR_dynamic_rendering` and `VK_KHR_synchronization2` extension aim to alleviate neuralgic pressure points of the API.



(a) *Assignment 1* **(b)** *Assignment 2* **(c)** *Assignment 3* **(d)** *Assignment 4* **(e)** *Assignment 5*

Figure 1: *Visual results of implementing all mandatory subtasks for each of the five different assignments in our course.*

3. Course Details and OpenGL Student Assignments

Our course "Introduction to Computer Graphics" targets bachelor/undergraduate students in their third semester and is usually their second encounter with rasterization-based graphics pipelines, but their first encounter with a graphics API. Rasterization is first introduced in the preceding course "Introduction to Visual Computing", where students have to manually implement selected parts of a rasterization pipeline such as polygon clipping and line rasterization.

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 [Ste22], GLFW [Löw21], and GLM [G-T21]), 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 in C/C++ for the host-side code across all five assignments. In addition, students need to implement approximately 200 LOC in GLSL shaders.

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 geometric objects as indexed triangle meshes (box, cylinder, sphere); loading the data onto the GPU into vertex buffer objects and creating vertex array objects; providing 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 [Pho75] combined with Gouraud shading [Gou71] and Phong shading [Pho75] 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

Table 1: *Course assignments and their subtasks*

4. Transitioning to Vulkan

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 credit hours (according to the European Credit Transfer and Accumulation System [Eu22]), we strove to create similar workloads for students of either route. Hence, 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 w.r.t. the Vulkan API and reducing implementation time. In the following, we describe which abstractions we ended up providing through the framework for each of the assignments.

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 [Khr22vsa]. 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.

```
// Instance, surface, physical and logical device,
// queue and swap chain config are prepared by
// students before calling the init function.
ecgInitFramework(inst, surf, phys, dev, q, swpcfg);

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

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

  ecgStartRecordingCommands();
  ecgDrawTeapot();
  ecgEndRecordingCommands();

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

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.*

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 "ecg" 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 the red teapot of Figure 1a 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`. 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 (Figure 1c 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 are shown in Figure 1b.

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

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

We refrain from introducing SPIR-V [Khr22spv], and from letting students compile shader modules on their own, but handle these parts internally in the framework using glslang [Khr22gsl]. 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 render 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.

```
VkBuffer ecgCreateHostCoherentBufferWithBackingMemory
    (VkDeviceSize, VkBufferUsageFlags);
void ecgCopyDataIntoHostCoherentBuffer(VkBuffer,
    const void*, size_t);
void ecgDestroyHostCoherentBufferAndItsBackingMemory(
    VkBuffer);
VkImage ecgCreateImageWithBackingMemory(uint32_t,
    uint32_t, VkFormat, VkImageUsageFlags);
void ecgDestroyImageAndItsBackingMemory(VkImage);
```

Listing 3: *Convenience functions for creating buffers and images, provided by the framework, which handle their associated device memory internally, opaquely to the user.*

For enabling depth testing in Assignment 2, a new image can be created through the function shown in Listing 3, specifying a suitable format and usage flags. Its handle must be provided via the custom swap chain configuration struct, as described in Listing 1.

In Assignment 3, one framework convenience functionality is removed, namely automatic command buffer recording. It was 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, as shown in Figure 1d) and uniform buffers for object data and light source data.

In Assignment 5, important 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 heavy-weight 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 shall be changed, that change can occur during render loop execution, which can lead to an expensive operation being performed within the render loop.

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 [Khr21vsp]. 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 wildly 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 [Hec18].

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.

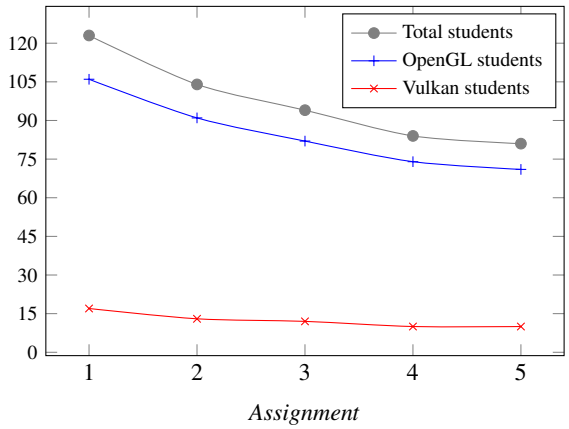


Figure 2: *Number of student submissions for all five assignments.*

6. Student Feedback

At the beginning of our university winter term in 2021, we offered all of our "Introduction to Computer Graphics" students the choice to choose between OpenGL and Vulkan. 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 the 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 Figure 2.

Figure 3 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 exact wording of our questions was as follows:

- How was the workload of Assignment X in your opinion? (Answers ranging from "There was 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/topics you picked up during Assignment X will be useful in your future career? (Answers ranging 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 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 Figure 2) for both groups.

The box plots in Figure 3 represent students' perceived workload, i.e., whether they felt that the necessary workload was appropriate for an assignment or not. The box plots in Figure 4 present students' estimates (or actual amounts) of hours spent on the mandatory tasks per assignment. It can be infered 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, while the minimum outliers point towards a slightly higher baseline in terms of required effort across all assignments for the Vulkan assignments. Interestingly, Vulkan students appeared to require 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, though, since they have chosen the Vulkan route despite our initial warning message about potentially higher workload, which we were unable to estimate beforehand. 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 our students' choice in favor of one API or the other.

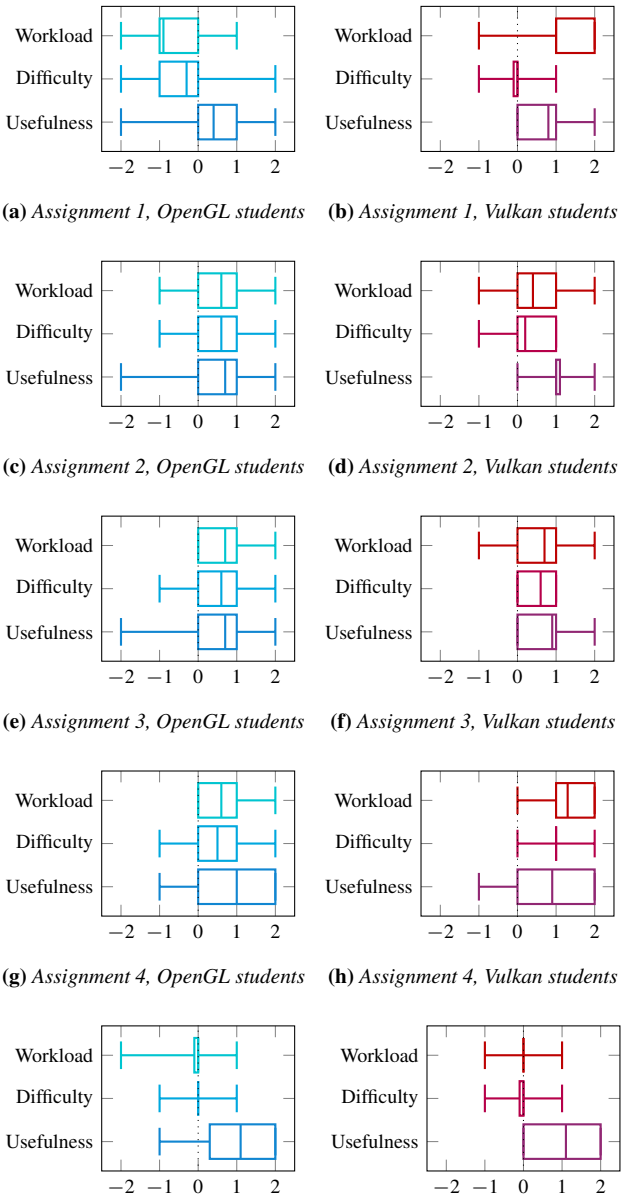
Reasons given by students for choosing Vulkan	**%**
Wanted to learn this API	100%
The provided framework seemed to be in a better state	33%
Vulkan is more relevant for game development	11%
More modern API	11%

Table 2: *Reasons of Vulkan students for choosing Vulkan. The second column states the percentage of Vulkan 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%
The 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%

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

With 60%, the strongest motivator of OpenGL students was to avoid the potentially higher workload of the Vulkan route. With only 38% of OpenGL students deliberately wanting to learn the OpenGL API, there is 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 at the 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



(a) *Assignment 1, OpenGL students* **(b)** *Assignment 1, Vulkan students*

(c) *Assignment 2, OpenGL students* **(d)** *Assignment 2, Vulkan students*

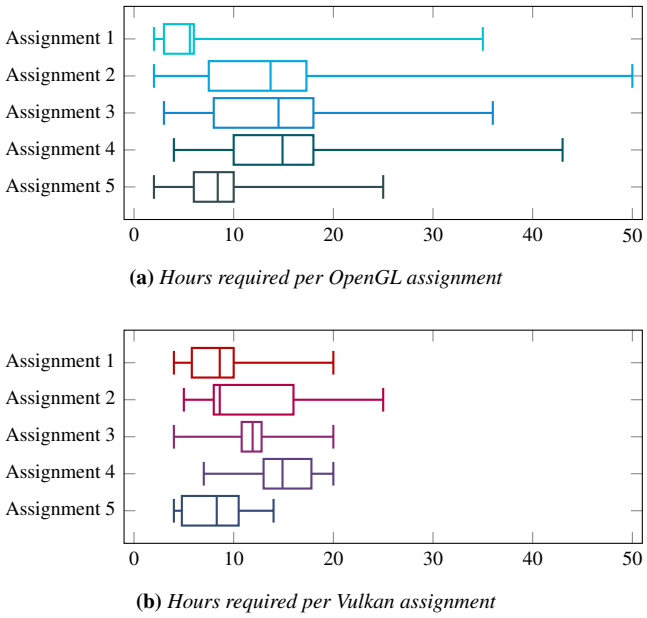
(e) *Assignment 3, OpenGL students* **(f)** *Assignment 3, Vulkan students*

(g) *Assignment 4, OpenGL students* **(h)** *Assignment 4, Vulkan students*

(i) *Assignment 5, OpenGL students* **(j)** *Assignment 5, Vulkan students*

Figure 3: *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.*

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 new Vulkan lectures and provided them to our



(a) *Hours required per OpenGL assignment*

(b) *Hours required per Vulkan assignment*

Figure 4: *Hours it takes to complete all mandatory tasks per assignment, as reported by students.*

students repeatedly throughout the course via the "Vulkan Lecture Series" [Tuw21]. Unfortunately, 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 leads to better and faster learning success.

Vulkan resource	**%**
vulkan-tutorial.com [Ove22]	89%
Vulkan specification [Khr21vsp]	78%
Assignment description documents	78%
"Vulkan Lecture Series" on YouTube [Tuw21]	44%
Official Khronos examples [Khr22vsa]	33%
Sascha Willems' tutorials and examples [Sas22t; Sas22e]	22%

Table 4: *Reported learning resources of Vulkan students.*

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 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 generally 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 report-

ing problems with framework usage (see Tables 2 and 3). Overall, Vulkan students stated to be pretty happy with the framework's level of abstraction (see Figure 5a). 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 Figure 3b). 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 Figure 5b).

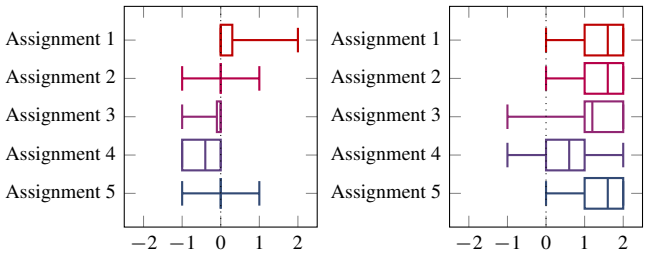
Problems with the OpenGL route	**%**
Graphics API usage (direct OpenGL API usage)	52%
Programming in C/C++	42%
Using the provided OpenGL framework	35%

Table 5: *Biggest problems of OpenGL 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%

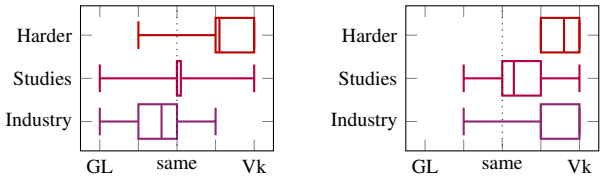
Table 6: *Biggest problems of Vulkan students during development.*

Figure 6 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. We plan to accommodate this request for the course framework in a similar way as we have added multi-platform support to our Vulkan frameworks Auto-Vk [Cg22avk] and Gears-Vk [Cg22gvk].



(a) *Level of abstraction* **(b)** *Learning experience*

Figure 5: *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.*



(a) *OpenGL students* **(b)** *Vulkan students*

Figure 6: *"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?"*

7. Conclusion

We successfully employed Vulkan for teaching the use of a real-time graphics API in an introductory course. 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. However, initial difficulty and workload were still rated higher, leading to a higher drop-out rate after Assignment 1 among Vulkan students. We plan to counteract this undesirable effect by providing more relevant learning resources early on and extending the period until the deadline by 1–2 weeks. Interestingly, the biggest hurdle for many students was C/C++ usage, constituting a bigger problem for Vulkan students since they had to write more code in Assignment 1. More efficient C/C++ learning resources and lectures should allow students to focus on graphics API usage. Difficulty and workload ratings of the other assignments converged mostly for both groups of students.

Our proposed assignment structure enables the step-wise introduction of increasingly challenging concepts, such as synchronization and image layout transitions, at a later point in the course. These concepts are still vital to students for becoming proficient users of modern graphics APIs, so students should get in touch with them at least briefly—even in an introductory graphics course. 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.

Acknowledgements

The original OpenGL framework was created by Lukas Gersthofer and Bernhard Steiner. We could build upon their great precedent work when creating the Vulkan counterpart. Lukas Gersthofer furthermore provided valuable feedback and helped to shape and improve the Vulkan framework's code through code reviews. We thank Hiroyuki Sakai for providing feedback and TikZ expertise. This work was supported by the Research Cluster "Smart Communities and Technologies (Smart CT)" at TU Wien and by the Austrian Science Fund (FWF), project no. F77.

References

[AMD18] AMD. *V-EZ API Documentation*. https://gpuopen-librariesandsdks.github.io/V-EZ/. [Accessed 22-January-2022]. 2018 2.

[Apl22] APPLE INC. *Metal. Accelerating graphics and much more*. https://developer.apple.com/metal. [Accessed 21-Mar-2022]. 2022 1.

[Bre22] THE BRENWILL WORKSHOP LTD. *KhronosGroup/MoltenVK*. https://github.com/KhronosGroup/MoltenVK. [Accessed 21-Mar-2022]. 2022 1.

[BWF17] BALREIRA, DENNIS GIOVANI, WALTER, MARCELO, and FELLNER, DIETER W. "What we are teaching in Introduction to Computer Graphics." *Eurographics (Education Papers)*. 2017, 1–7 2.

[Cg22avk] RESEARCH UNIT OF COMPUTER GRAPHICS | TU WIEN. *Auto-Vk: Low-level convenience and productivity layer atop Vulkan-Hpp*. https://github.com/cg-tuwien/Auto-Vk. [Accessed 21-Mar-2022]. 2022 8.

[Cg22gvk] RESEARCH UNIT OF COMPUTER GRAPHICS | TU WIEN. *Gears-Vk: Powerful low-level C++20 rendering framework for Vulkan 1.2, including Real-Time Ray Tracing (RTX) support, built atop Auto-Vk.* https://github.com/cg-tuwien/Gears-Vk. [Accessed 21-Mar-2022]. 2022 8.

[CXR18] CHEN, MINSI, XU, ZHIJIE, and RIPPIN, WAYNE. "On the Pedagogy of Teaching Introductory Computer Graphics without Rendering API". *Eurographics technical report series EG* 2018 (2018), 47–50 2.

[dVri22] DE VRIES, JOEY. *LearnOpenGL - Hello Triangle*. https://learnopengl.com/Getting-started/Hello-Triangle. [Accessed 22-January-2022]. 2022 2.

[Eu22] EUROPEAN UNION. *European Credit Transfer and Accumulation System (ECTS)*. https://education.ec.europa.eu/levels/higher-education/inclusion-connectivity/european-credit-transfer-accumulation-system. [Accessed 21-Mar-2022]. 2022 3.

[FWW12] FINK, HEINRICH, WEBER, THOMAS, and WIMMER, MICHAEL. "Teaching a modern graphics pipeline using a shader-based software renderer". *Computers & Graphics* 37.1-2 (2012), 12–20 2.

[G-T21] G-TRUC CREATION. *OpenGL Mathematics (GLM)*. https://github.com/g-truc/glm. [Accessed 22-Jan-2022]. 2021 3.

[Gou71] GOURAUD, HENRI. "Continuous shading of curved surfaces". *IEEE transactions on computers* 100.6 (1971), 623–629 3.

[Hec18] HECTOR, TOBIAS. *Comparing the Vulkan SPIR-V memory model to C++'s*. https://www.khronos.org/blog/comparing-the-vulkan-spir-v-memory-model-to-cs. [Accessed 21-Mar-2022]. 2018 5.

[Khr18ray] THE KHRONOS® GROUP INC. *VK_NV_ray_tracing(3) Manual Page*. https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_NV_ray_tracing.html. [Accessed 21-Mar-2022]. 2018 2.

[Khr20ray] THE KHRONOS® GROUP INC. *VK_KHR_ray_tracing_pipeline(3) Manual Page*. https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_KHR_ray_tracing_pipeline.html. [Accessed 21-Mar-2022]. 2020 2.

[Khr21vsp] THE KHRONOS® GROUP INC. *Vulkan® 1.2.203 - A Specification (with all registered Vulkan extensions)*. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html. [Accessed 24-Jan-2022]. 2021 5, 7.

[Khr22gsl] THE KHRONOS® GROUP INC. *KhronosGroup/glslang: Khronos-reference front end for GLSL/ESSL, partial front end for HLSL, and a SPIR-V generator.* https://github.com/KhronosGroup/glslang. [Accessed 24-Jan-2022]. 2022 4.

[Khr22me] THE KHRONOS® GROUP INC. *Khronos Members - The Khronos Group Inc*. https://www.khronos.org/members/list. [Accessed 21-Mar-2022]. 2022 1.

[Khr22spv] THE KHRONOS® GROUP INC. *SPIR Overview - The Khronos Group Inc*. https://www.khronos.org/spir. [Accessed 24-Jan-2022]. 2022 4.

[Khr22vk] THE KHRONOS® GROUP INC. *Vulkan | Cross platform 3D Graphics*. https://www.vulkan.org. [Accessed 21-Jan-2022]. 2022 1.

[Khr22vsa] THE KHRONOS® GROUP INC. *KhronosGroup/Vulkan-Samples: One stop solution for all Vulkan samples*. https://github.com/KhronosGroup/Vulkan-Samples. [Accessed 24-Jan-2022]. 2022 4, 7.

[Löw21] LÖWY, CAMILLA. *An OpenGL Library | GLFW*. https://www.glfw.org/. [Accessed 22-Jan-2022]. 2021 3.

[Msf22] MICROSOFT CORPORATION. *DirectX graphics and gaming*. https://docs.microsoft.com/en-us/windows/win32/directx. [Accessed 21-Mar-2022]. 2022 1.

[Ove22] OVERVOORDE, ALEXANDER. *Vulkan Tutorial*. https://vulkan-tutorial.com. [Accessed 22-January-2022]. 2022 2, 7.

[Pho75] PHONG, BUI TUONG. "Illumination for computer generated pictures". *Communications of the ACM* 18.6 (1975), 311–317 3.

[RME14] REINA, GUIDO, MÜLLER, THOMAS, and ERTL, THOMAS. "Incorporating modern opengl into computer graphics education". *IEEE computer graphics and applications* 34.4 (2014), 16–21 2.

[Sas22e] WILLEMS, SASCHA. *SaschaWillems/Vulkan: Examples and demos for the new Vulkan API*. https://github.com/SaschaWillems/Vulkan. [Accessed 16-Mar-2022]. 2022 7.

[Sas22t] WILLEMS, SASCHA. *Vulkan Tutorial - Sascha Willems*. https://www.saschawillems.de/tags/vulkan-tutorial/. [Accessed 16-Mar-2022]. 2022 7.

[SSKL13] SHREINER, DAVE, SELLERS, GRAHAM, KESSENICH, JOHN M., and LICEA-KANE, BILL M. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. 8th. Addison-Wesley Professional, 2013. ISBN: 0321773039 2.

[Ste22] STEWART, NIGEL. *GLEW - The OpenGL Extension Wrangler Library*. https://github.com/nigels-com/glew. [Accessed 16-Mar-2022]. 2022 3.

[SWH15] SELLERS, GRAHAM, WRIGHT, RICHARD S., and HAEMEL, NICHOLAS. *OpenGL Superbible: Comprehensive Tutorial and Reference*. 7th. Addison-Wesley Professional, 2015. ISBN: 0672337479 2.

[SWL17] SUSELO, THOMAS, WÜNSCHE, BURKHARD C, and LUXTON-REILLY, ANDREW. "The journey to improve teaching computer graphics: A systematic review". *Proc. 25th Int. Conf. Comput. Educ.* 2017, 361–366 2.

[Tuw21] TU WIEN, INSTITUTE OF VISUAL COMPUTING AND HUMAN-CENTERED TECHNOLOGY. *Vulkan Lecture Series - Computer Graphics at TU Wien*. https://www.youtube.com/watch?v=tLwbj9qys18&list=PLmIqTlJ6KsE1Jx5HV4sd2jOe3V1KMHHgn. 2021 7.

[UKPW21] UNTERGUGGENBERGER, JOHANNES, KERBL, BERNHARD, PERNSTEINER, JAKOB, and WIMMER, MICHAEL. "Conservative Meshlet Bounds for Robust Culling of Skinned Meshes". *Computer Graphics Forum* 40.7 (Oct. 2021), 57–69. ISSN: 1467-8659. DOI: 10.1111/cgf.14401. URL: https://www.cg.tuwien.ac.at/research/publications/2021/unterguggenberger-2021-msh/ 1.

[UKS*20] UNTERGUGGENBERGER, JOHANNES, KERBL, BERNHARD, STEINBERGER, MARKUS, et al. "Fast Multi-View Rendering for Real-Time Applications". *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by FREY, STEFFEN, HUANG, JIAN, and SADLO, FILIP. Eurographics. online, May 2020, 13–23. ISBN: 978-3-03868-107-6. DOI: 10.2312/pgv.20201071. URL: https://www.cg.tuwien.ac.at/research/publications/2020/unterguggenberger-2020-fmvr/ 1.