Eurographics Symposium on Parallel Graphics and Visualization (2020)
S. Frey, J. Huang, F. Sadlo (Editors)

Fast Multi-View Rendering for Real-Time Applications

Johannes Unterguggenberger[1], Bernhard Kerbl[1], Markus Steinberger[2], Dieter Schmalstieg[2], and Michael Wimmer[1]

[1]TU Wien, Institute of Visual Computing & Human-Centered Technology, Austria
[2]Graz University of Technology, Austria

Abstract

Efficient rendering of multiple views can be a critical performance factor for real-time rendering applications. Generating more than one view multiplies the amount of rendered geometry, which can cause a huge performance impact. Minimizing that impact has been a target of previous research and GPU manufacturers, who have started to equip devices with dedicated acceleration units. However, vendor-specific acceleration is not the only option to increase multi-view rendering (MVR) performance. Available graphics API features, shader stages and optimizations can be exploited for improved MVR performance, while generally offering more versatile pipeline configurations, including the preservation of custom tessellation and geometry shaders. In this paper, we present an exhaustive evaluation of MVR pipelines available on modern GPUs. We provide a detailed analysis of previous techniques, hardware-accelerated MVR and propose a novel method, leading to the creation of an MVR catalogue. Our analyses cover three distinct applications to help gain clarity on overall MVR performance characteristics. Our interpretation of the observed results provides a guideline for selecting the most appropriate one for various use cases on different GPU architectures.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Rasterization;*** *Visibility; Virtual reality;*

1. Introduction

Consumer-grade head-mounted displays (HMDs) have become popular for virtual reality (VR) in recent years, and new VR games are being released regularly. At the time of writing, Valve's Steam Store already lists more than 4000 games tagged as "VR Only" [Val03]. Inherent to VR games are increased requirements on the rendering performance of a PC or gaming console because every frame has to be rendered *at least* twice—i.e., at least once for each eye—with view positions slightly offset. However, two views might not be sufficient for an HMD with a wide field of view and non-coplanar displays [BS18]. Four or more views can be required for such setups.

Efficient rendering of multiple views does not only have its applications in VR rendering or in rendering for multi-monitor/multi-projector setups. Multiple ID buffers containing primitive IDs can be evaluated in order to determine which primitives are visible from a range of viewpoints, i.e., a *potentially visible set* (PVS). Such a PVS can be used to e.g. shade all triangles which may become visible under head movement [MVD*18]. Another application scenario is shadow mapping for multiple light sources. Each light source represents the origin of at least one view frustum that corresponds to the region that is illuminated by that light. For omnidirectional lights and algorithms like cascaded shadow mapping [Dim07], multiple views must be rendered per light source.

Producing multiple views per frame while maintaining frame rates of at least $60Hz$ can be challenging. Rendering effort depends heavily on the scene representation and the graphics processing unit (GPU) which renders the scene. For real-time VR applications, usually, the requirements are even higher. Vlachos [Vla16] recommends staying below $10ms$ time per frame to achieve stable frame rates at $90Hz$. In general, it can be stated that there is a need for multi-view rendering (MVR) techniques which enable efficient processing of several viewpoints and are versatile enough to be used for arbitrary scene setups and across different GPUs.

Hardware manufacturers such as Oculus [Eve16] and NVIDIA [NVI18b] have shown increased interest in the efficient rendering of multiple views. Hardware-accelerated MVR is commonly exposed as an extension for existing graphics APIs. For OpenGL and OpenGL ES, the extension is called *OVR_multiview* [Cas18] and has been implemented by NVIDIA, ARM, Qualcomm, and Imagination Technologies. Hardware-accelerated MVR is likely to outperform any other approach for MVR, including software techniques that make intelligent use of shader programs to minimize the number of draw calls and memory transfer [Wil15, DNS10].

However, to the best of our knowledge, there is little information available that lets one quantify the actual benefits of using one MVR method over another on recent GPU models. An additional caveat for desktop systems is that hardware-accelerated MVR on consumer-grade NVIDIA GPUs is limited in its applicability to real-time graphics, due to its lack of geometry shader and tessellation shader support [Wil16]. A detailed analysis of available MVR methods with modern graphics APIs would enable developers to make informed choices in their design without resorting to a trial-and-error process.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/pgv.20201071

To gain clarity about the performance of hardware-accelerated MVR and software-based methods for GPUs, we provide an exhaustive evaluation of various techniques for rendering with multiple viewpoints in different scenes on a range of recent GPU models, for three distinct MVR applications. Specifically, we evaluate relevant MVR methods in the context of ID buffer generation for PVS, lightfield G-buffer rendering and shadow mapping. For our performance tests, we implement and test more than 50 different rasterization pipeline configurations, including the techniques of Sorbier et al. [DNS10], Wilson [Wil15], Vlachos [Vla15], different variants of hardware-accelerated MVR pipelines, and altogether new variants. To facilitate the identification and imparting of individual methods, we introduce a formalized syntax to describe custom MVR pipelines. In summary, our contributions include the following:

- We introduce a symbol-based description language to declare specific pipeline configurations for MVR in a concise manner.
- We examine the emergent performance characteristics of available hardware-accelerated MVR and compare them to other pipeline variants, including previously published techniques.
- We analyze and interpret performance trends for the most relevant MVR pipeline variants across different GPUs and scenes. In comparison to previous work, we also consider much larger configurations with up to 32 simultaneously rendered views.
- We describe two optimized, geometry shader-based MVR variants and identify applications where they can be used as viable alternatives to hardware-accelerated MVR. In contrast to the latter, these general variants preserve full support for custom tessellation and geometry shader routines on consumer-grade devices.

In the following, we summarize related work and previous efforts to achieve efficient MVR in hardware and software (Section 2). In Section 3, we introduce our symbol-based parameter syntax for describing different pipeline variants that are suitable for MVR. Our setup and full evaluation, along with obtained results, are described in Section 4. We analyze emergent performance trends and give interpretations, as well as additional important insights in Section 5. Finally, we provide a summary and outlook in Section 6.

2. Related Work

A considerable body of previous work has addressed the problem of multi-view rendering in computer graphics. As long as view positions only change in terms of rotation, textured impostor rectangles can be used as stand-ins for actually transformed scene geometry [SS96]. Halle et al. present an alternative scene representation and rendering algorithm that enable significant speedup of view-dependent computations by enforcing restrictions w.r.t. the discrepancies between views [Hal98]. Specifically, all camera positions must lie on a single translational axis along which views can be sampled. Sitthi-Armon et al. [SaLY*08] describe how to make use of reprojection to avoid shading computations for slightly differing views by using cached results from previous frames. A particular application of *decoupled sampling* is the smooth generation of visibility for multiple new views, yielding superior results to caching approaches [RKLC*11].

Beyond straightforward implementations, there are several aspects of image synthesis with MVR that bear potential for optimization. Adelson et al. [ABC*91] provide a detailed analysis on this topic in the context of stereoscopic projections. The authors propose several methods to avoid duplicate attribute computations, efficiently cull geometry that is invisible to both eyes and resolve visibility by combining Z-buffers with BSP trees for depth testing. Based on these ideas, several methods and mechanisms for modern graphics APIs have been proposed to improve the performance of MVR over simple multi-pass rendering. Marbach [Mar09], as well as Beck et al. [BSF10], provide basic evaluations on the benefits of geometry shaders and layered rendering for MVR, with mixed results. The techniques of Marbach [Mar09] and Sorbier et al. [DNS10] have in common that they aim to reduce driver overhead and increase GPU utilization by supplying all active views with a single draw call: rendered geometry is amplified in a geometry shader loop. Each view's pixel values are written to a separate layer of an array texture [Mar09] or to an exclusive region in a single texture, where the single texture contains all views to be rendered [DNS10]. An aspect of the technique by Sorbier et al. is that culling and clipping cannot be performed implicitly by the rasterizer, which the authors address by discarding all "out of bounds" writes in the fragment shader.

A more recent approach by Wilson [Wil15] also relies on the single-texture approach, but uses instanced rendering to achieve geometry amplification. Furthermore, they define custom clip planes in the vertex shader to avoid out of bounds writes, thus saving on potentially expensive fragment discards. To achieve efficient MVR with point type primitives, Marrs et al. [MWH18] avoid the rendering pipeline altogether and use compute shaders instead. Unfortunately, previous work on software GPU rasterizers has shown that similar performance gains cannot be expected for triangle meshes [KKSS18].

Hasselgren et al. [HAM06] have conceived and simulated their prototype of a complete VR-oriented architecture that aims to maximize exploitation of coherence between views. Starting with the Pascal microarchitecture, NVIDIA has added built-in hardware support for MVR that is exposed in VRWorks [NVI18b] and OpenGL by the Oculus Virtual Reality (OVR) multi-view extension [Cas18]. Driven by the need for fast stereoscopic projection in VR, the *Single Pass Stereo* functionality optimizes rendering to two separate viewpoints. With the Turing microarchitecture, NVIDIA has further expanded on this feature set by adding support for accelerated rendering of up to four separate viewpoints [NVI18a].

Recently, streaming rendering techniques for VR have been proposed [MVD*18, HSS19b]. Inspired by early work on optimizing VR application pioneered by Regan et al. [RP94], these approaches require the computation of a potentially visible set (PVS) of geometry to be shaded on a server and then streamed for framerate upsampling to a client, e.g., a head-mounted display. For PVS computations, these approaches render four to eight frames along the predicted head movement, leading to a typical MVR problem: generating multiple primitive ID buffers quickly. As an alternative to sampled visibility, Hladky et.al [HSS19a] proposed a conservative single pass PVS computation. While this avoids MVR, it requires up to a hundred ms for typical scenes, raising the question of whether efficient MVR rendering may not be a better solution to the problem.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

3. Classification

In order to exhaustively analyze the properties of different MVR techniques and discuss their mechanics, we first establish a method classification catalogue that enables us to capture all relevant properties with a compact, intuitive representation. To this end, we introduce a formal notation to represent an arbitrary MVR technique that processes N different views as a pipeline function $\mathcal{P}(\dots)$ whose parameters define its implementation specifics. We propose a parameter set that is based on the variety of pipelines presented in previous work, as well as additional attributes that we found to facilitate their classification in practice during our experiments. In our current model, we consider four essential properties:

Pipeline invocation count The number of times the pipeline must be run from start to finish in order to process all N views.
Geometry amplification The mechanism used for producing sufficient copies of the input geometry to provide each view.
Custom culling Required or supplemental steps included in the pipeline to perform culling and/or clipping of triangle primitives.
Framebuffer Layout The layout and configuration for the framebuffer object that the fragment shader writes its output to.

In the following, we elaborate on the significance of each individual parameter and its effects on technique configuration. In addition, we provide illustrations of selected pipeline examples.

List of Symbols

Geometry Amplification

Symbol	Meaning
⟶	Direct forwarding
⋿	Amplification by instanced rendering
⊢≪	Amplification by geometry shader loop
⊢⋿	Amplification by geometry shader instancing
≋	(Accelerated) OVR geometry amplification

Framebuffer Layout

Symbol	Meaning
□‥□	Separate framebuffer objects
⊞	Single large, partitioned framebuffer
⧉	Layered framebuffer
⊞‥⊞	Multiple partitioned framebuffers
⧉‥⧉	Multiple layered framebuffers

Culling & Clipping

Symbol	Meaning
$\texttt{CLIP}_{VP}$	Clipping with reduced viewport
$\texttt{CLIP}_{\|\|}$	Clipping with clip planes
$\texttt{CLIP}_{FS}$	Clipping in fragment shader
$\texttt{VFC}_{GS}$	Frustum culling in geometry shader
$\texttt{BFC}_{GS}$	Backface culling in geometry shader

3.1. Pipeline Invocation Count

When executing an MVR pipeline, it may be desired or necessary to run the entire pipeline multiple times. Let us consider the most straightforward method to achieve MVR, which is to invoke multiple draw calls that write the result for each of the N different views to a separate target texture. In this case, the graphics pipeline must run from start to end N times per scene entity to produce N views. Figure 1a illustrates this basic pipeline setup and the necessary steps for each invocation.

Running the full pipeline multiple times may also be required to circumvent hardware restrictions for specific techniques. For instance, the OVR extension enables efficient hardware acceleration on NVIDIA Turing models only when using four target views or fewer [NVI18a]. One way to evaluate hardware-accelerated MVR for a larger number of views is thus to split the N views into groups of four and invoke the entire pipeline multiple times. We indicate such an approach by setting the first parameter of a pipeline function to $\lceil \frac{N}{4} \rceil$. On the NVIDIA Pascal microarchitecture, only two views can be hardware-accelerated [NVI16], therefore, e.g., a test with $\lceil \frac{N}{2} \rceil$ invocations would be of particular interest in such a scenario.

3.2. Geometry Amplification

The key requirement for achieving MVR is the amplification of the input geometry, cuing the rasterizer to render multiple instances of each primitive—once for each view. Modern graphics APIs offer various ways to achieve this amplification at different access points in the pipeline. The choice of access point affects the quantity and nature of work that the GPU must handle. The earlier amplification occurs, the more stages must process an amplified amount of data.

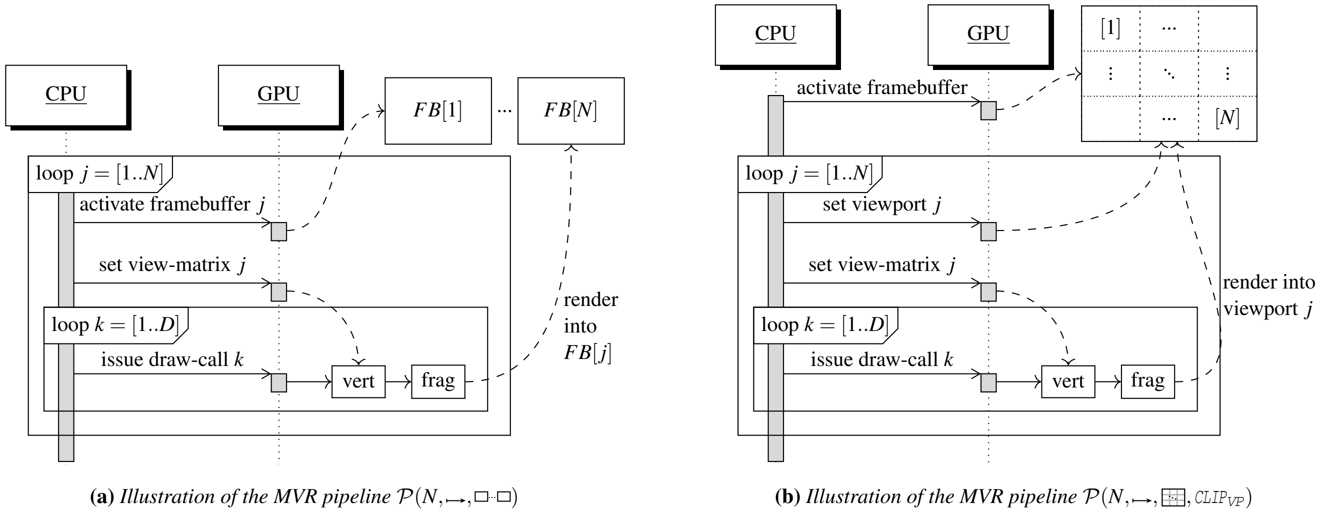
At its earliest, geometry amplification can be done at the very beginning of a rasterization pipeline, which effectively means invoking the pipeline multiple times, each time with a different viewpoint location. In this case, all geometry that enters the pipeline is simply forwarded. We indicate this behaviour by the ⟶ symbol in the pipeline's geometry amplification parameter field. The simple pipelines in Figure 1 both use this setting. In order to reduce the number of draw calls without losing any flexibility, API calls that perform *instanced rendering* (signified by ⋿) can be used instead.

In the case of MVR, we are only interested in duplicating view-dependent computations. By moving the amplification to the end of the geometry stage, we can thus avoid redundant invocations of vertex shaders that, e.g., compute skeletal animation, which is uniform across all views in a given frame. In the geometry shader, output primitives can either be emitted in a loop (⊢≪) or, if the number of duplicates is fixed, via geometry shader instancing (⊢⋿).

Finally, specific extensions for MVR have been added to rendering APIs. A powerful example is the OVR extension: on modern NVIDIA GPUs, it enables hardware-accelerated geometry amplification, which exploits re-usability of shading results across multiple views. On architectures that have no built-in support, the OVR functionality will usually fall back to a looping behaviour. We indicate this (flexible) type of geometry amplification with the ≋ symbol. The declared target application for this functionality is stereoscopic rendering for VR, where the majority of geometry computations and visibility tests are valid for both eyes [NVI18b]. Unfortunately, using the OVR extension also prohibits the use of any custom tessellation or geometry shaders on all consumer-grade GPUs [Wil16].

Note that for all amplification methods, the number of copies generated is implicitly given by the number of times a pipeline is run. If it is called only once, geometry amplification must generate N copies. For $\lceil \frac{N}{4} \rceil$, each run must amplify the input geometry by a factor of up to $\times 4$. A pipeline that does not generate sufficient geometry for all N desired views is not valid in our definition.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.



(a) *Illustration of the MVR pipeline* $\mathcal{P}(N, \hookrightarrow, □⋯□)$ **(b)** *Illustration of the MVR pipeline* $\mathcal{P}(N, \hookrightarrow, ⊞, \texttt{CLIP}_{\text{VP}})$

Figure 1: *Examples of MVR configurations corresponding to our definition syntax. (a) A straightforward MVR pipeline uses N invocations to write each view into a different framebuffer. (b) A multi-pass variant with a single, partitioned framebuffer and varying viewports for clipping.*

3.3. Custom Culling and Clipping

Some MVR techniques require additional steps after geometry amplification and before storing each view's result into its target framebuffer. When using a partitioned framebuffer, i.e., a framebuffer which contains multiple views, we must ensure that triangles are adequately clipped against the current target region and do not protrude into regions that correspond to a different view. This can be achieved in several ways: For one, graphics APIs provide methods for reconfiguring the viewport against which culling and clipping are performed between draw calls. We indicate that this feature is being used by $\texttt{CLIP}_{\text{VP}}$. Alternatively, we can avoid this additional API command and implied synchronization dependencies by defining custom clip planes ($\texttt{CLIP}_{||}$) in the vertex shader instead [Wil15]. A third method exploits the `discard` instruction in the fragment shader ($\texttt{CLIP}_{\text{FS}}$) to achieve correct clipping [DNS10]. In addition to purely functional clipping, we also consider the impacts of performing fine-grained view frustum culling and backface culling in the geometry shader to reduce its output, which we denote with the symbols $\texttt{VFC}_{\text{GS}}$ and $\texttt{BFC}_{\text{GS}}$, respectively. If multiple methods are used in the same pipeline, they are concatenated by the | symbol.

3.4. Framebuffer Layout

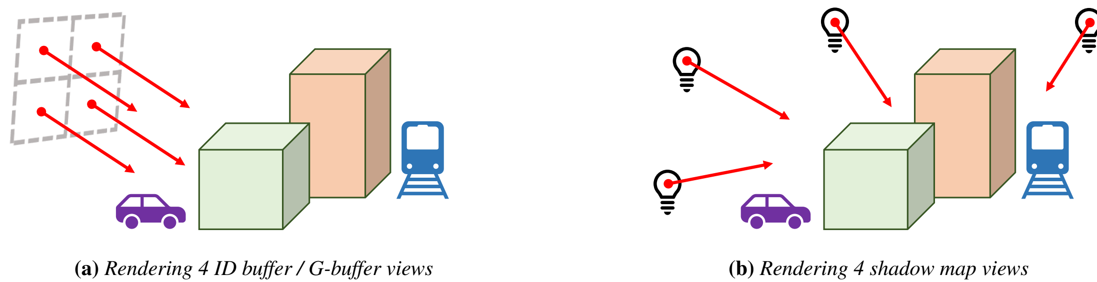
There are several possible choices w.r.t. the layout for storing the collective results generated for all N processed views. In our cases, we consider all framebuffer objects to contain at least one depth buffer and an arbitrary number of colour targets. In the simplest case, N separate framebuffer objects and associated textures are allocated and each one is bound directly before rendering a particular view. We indicate this layout with the □⋯□ symbol, which is also used to describe the simple pipeline setup in Figure 1a. Note that this layout is extremely restrictive, as it implies that no API- or hardware-backed geometry amplification can be used, since framebuffer bindings cannot be changed while a graphics pipeline runs. Consequently, a common suggestion in previous work is to use a single large framebuffer object instead of multiple smaller ones, and specifying the target write window for each individual view [DNS10, Wil15]. We use the ⊞ symbol to represent such a pipeline configuration. Since ⊞ pipeline configurations eventually contain multiple view-results in one single framebuffer, one of the clipping methods described in Chapter 3.3 is indispensable for producing correct results. A third option for the framebuffer layout is to exploit layered rendering capabilities to write each view's content to a separate layer of an array texture [Mar09]. When an array texture is part of the framebuffer object, the rasterizer is executed in a special mode that allows setting the built-in layer ID that each primitive is assigned to. A layered framebuffer is indicated by the ⧉ symbol.

We have already noted in Chapter 3.1 that there are cases where we would like to partition the rendering of N views into chunks of, e.g., four views each and, as a consequence, issue $\lceil \frac{N}{4} \rceil$ draw calls per scene entity to produce the entirety of N view results. For such variants, we are using either multiple separate framebuffer objects of the principal ⊞ type, or multiple separate framebuffer objects of the principal ⧉ type. To indicate a set of conjugate ⊞ framebuffer objects, we use the ⊞⋯⊞ symbol. To indicate a set of conjugate ⧉ framebuffer objects, we use the ⧉⋯⧉ symbol.

4. Evaluation

In order to thoroughly evaluate and identify the conditions that influence an MVR technique's performance, we have collected timing results for several scenes, GPU models and setup configurations in three different applications. Specifically, we have evaluated more than 50 MVR variants in the context of ID buffer rendering for PVS computation, G-buffer generation for lightfields and shadow mapping (see Figure 2). For rendering a single ID buffer, we use multiple viewpoints on the surface of a rectangle to sample the scene visibility. For the lightfield G-buffer, we use the same sample positions as for ID buffer rendering but set multiple colour targets to store all fragment shader outputs. With shadow mapping, the discrepancy

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.



(a) *Rendering 4 ID buffer / G-buffer views* **(b)** *Rendering 4 shadow map views*

Figure 2: *Different configurations for a single 4-view setup in our applications. (a) To generate ID buffers for a PVS, we sample a rectangle to obtain visibility information under small camera motion as done, e.g., by [MVD*18]. The same samples can be used for generating a G-buffer for a small lightfield. (b) For shadow maps, samples are arbitrarily distributed, since light source positions are generally independent.*

between viewpoints in MVR is random, since the positions of light sources in a scene are mostly independent. We evaluate our MVR applications at up to 100 PVS/light source origins and orientations which are uniformly distributed in each scene. Each application's run time is recorded for a varying number of target views in 6 scenes listed in Table 1. In contrast to most previous work, our evaluation also considers large MVR setups and ranges from 2 to 32 simultaneous target views. We consider two common framebuffer resolutions for LQ/HQ purposes: 800×600 and $1080p$.

For our evaluation, we have implemented a testbed that enables users to quickly define and run a wide range of MVR techniques. We have chosen to use OpenGL 4.6 over Vulkan as the target rendering API, for two reasons: first, since Vulkan has not yet fully penetrated the industry, results obtained with OpenGL better reflect the expected impact in current graphics applications. Second, several helpful tools that allow for in-depth analysis (such as the Nsight Graphics range profiler) are currently incompatible with the Vulkan API [NVI20]. Furthermore, given that our test applications do not show high CPU workload or rely on complex input resource management, we expect deviations to be minor. We have recorded our results for the following GPUs: NVIDIA's GTX 980, GTX 1060, GTX 1650 SUPER, RTX 2080, RTX 2800 Ti and AMD's RX 580. Scenes undergo CPU-side frustum culling and have backface culling enabled in the rendering API. While we performed the measurements on different machines, our timings record only the portion of the GPU-side frame time required for rendering all N views. Before timing, we ran a warmup phase of 15 frames. For a single measurement, we uploaded the required resources to the GPU, waited for completion of previous commands, and recorded the multi-view rendering time using GPU timer queries. For evaluation, we consider the average frame times across all measurements per configuration. Due to the vast size of the parameter space for this problem, we must restrict our evaluation to cases that are of particular interest. In the following, we first identify a subset of most robust techniques out of the possible combinations that result from the parameters in Section 3. We consider ID buffer rendering as our base use case, since it requires minimal effort w.r.t. to vertex and fragment shading (i.e., it generates a single integer output) and should enable hardware-accelerated techniques to achieve peak performance due to the strong correlation of visible geometry between views [BS18]. This subset of techniques is then further refined to yield the most promising ones, for which we analyze trends and explore the impact of changing load parameters by applying them to G-buffer rendering (three vector-valued outputs) and shadow mapping (depth-only).

Table 1: *Scenes used to generate test results, along with geometry properties and the usual range of draw calls needed to produce one view of each scene. Due to frustum culling, the number of draw calls varies depending on the active view and pipeline configuration.*

	#vertices	#triangles	avg. #drawcalls
Bistro	2.52M	2.83M	43–71
Gallery	0.65M	1.00M	29–48
Robot Lab	0.38M	0.47M	46–111
San Miguel	9.02M	9.98M	866–1446
Sponza	0.29M	0.44M	84–137
Viking Village	2.87M	4.26M	195–384

4.1. Identifying Robust MVR Techniques

For the sake of brevity, we do not include results for all techniques that were part of our initial experiments. Instead, we provide a comparison of techniques that are based on previous work, as well as hardware-accelerated variants and promising combinations of API features that have not been proposed before. In order to identify the top-performing pipeline variants per category (where the categories are described in Sections 4.1.1 to 4.1.3), we compare them to the simplest possible MVR baseline: $\mathcal{P}(N, \mapsto, \square\cdots\square)$, which describes a pipeline that requires N pipeline invocations, performs simple geometry forwarding, and renders the results into separate framebuffer objects. A technique is considered robust on a given GPU if it performs faster than our baseline in at least 50% of all ID buffer rendering setups, which include different scenes, view counts and framebuffer resolutions. Note that we skip this comparison for the AMD RX580 entirely; no MVR technique performed significantly better than $\mathcal{P}(N, \mapsto, \square\cdots\square)$.

4.1.1. Instanced Rendering

Based on the method proposed by Wilson [Wil15], $\mathcal{P}(1, \rightrightarrows, \boxplus, \texttt{CLIP}_{||})$ describes an MVR pipeline which uses a single pass and instanced rendering for geometry amplification, forwarding the output to a large, partitioned framebuffer. To avoid

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

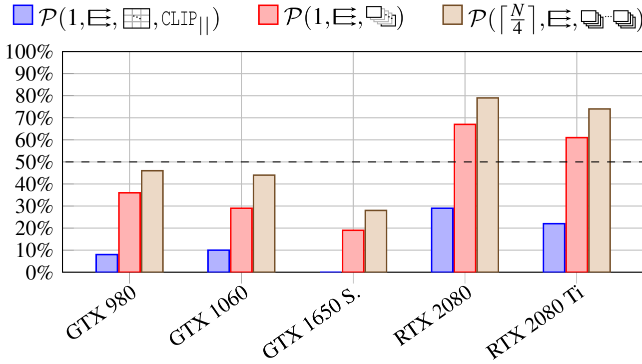


Figure 3: *Bars represent the percentage of configurations where a particular instanced rendering variant showed noticeably better performance than $\mathcal{P}(N, \rightarrow, \square\square)$ did for the same configuration.*

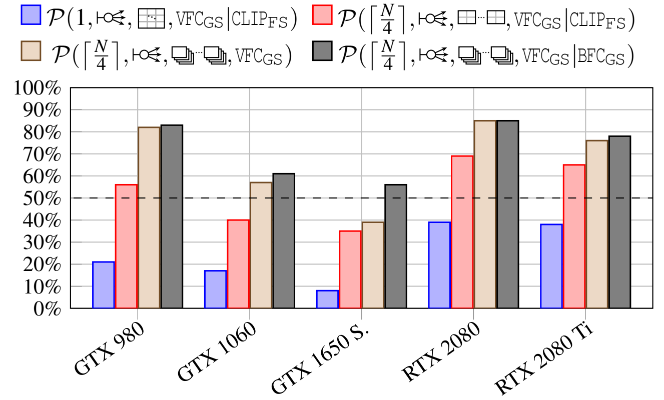


Figure 4: *Percentages of tests where each geometry shader-based variant showed performance improvements over our baseline.*

writing outside each view's bounds, custom vertex clip planes are used, which allows skipping the geometry shader stage altogether. While testing this approach in our setup, we found that its overall performance can be improved by using a layered framebuffer instead of a partitioned one. Note that in this case, we must set the layer ID for each primitive, which theoretically requires the presence of a geometry shader. However, for such constant-time efforts, modern NVIDIA GPUs support *pass-through* geometry shaders, which almost completely avoid the overhead caused by this stage. The $\mathcal{P}(1, ⇶, \square)$ variant employs this particular setup and consistently outperforms the original version on all tested NVIDIA GPUs—in most cases even by a large margin. We further found that the performance of this approach can be improved by restricting the number of views that are rendered at the same time. Our empirical tests have shown that limiting the number of simultaneously processed views to 4 works best, which will be a recurring theme in the following sections. An interpretation of this trend will be provided in Section 5. The resulting instanced rendering-based pipeline variant is described with the symbol $\mathcal{P}(\lceil \frac{N}{4} \rceil, ⇶, \square\square)$ and shows the most favourable ratio across the pipeline variants considered in this section when compared to our baseline (see Figure 3). Since none of these variants performed better than our baseline on weaker GPU models for 4 or more views, they are excluded from our detailed analysis in this paper. Corresponding results can be found in our supplemental material.

4.1.2. Geometry Shader-Based Techniques

Like Wilson's approach, the approach by Sorbier et al. [DNS10] targets a single, partitioned framebuffer and uses a single invocation to produce N views. However, their geometry amplification occurs in a geometry shader loop. The geometry shader further applies frustum culling to reduce the input to the rasterizer and performs clipping in the fragment shader to restrict rendering to each view's framebuffer region. Their pipeline variant can thus be denoted by $\mathcal{P}(1, \mapsto, \boxplus, \texttt{VFC}_{\texttt{GS}}|\texttt{CLIP}_{\texttt{FS}})$. Once again, restricting the number of simultaneous views provides a significant performance boost. We further saw that switching the large framebuffers for layered ones yields overall better performance. This is partly due to the fact that the fragment shader stage must no longer perform `discard` operations to achieve clipping and can take advantage of early depth testing. However, we also found that keeping the frustum culling routine in the geometry shader is beneficial; since the geometry shader is executed in a loop, testing each triangle against a frustum incurs only a small overhead which can be amortized by the reduced output of the geometry shader. We denote this pipeline as $\mathcal{P}(\lceil \frac{N}{4} \rceil, \mapsto, \square\square, \texttt{VFC}_{\texttt{GS}})$. To further relieve the rasterization stage, we have added backface culling to the geometry shader routine, which led to a relative speedup in more than 60% of all cases. The advantageousness of choosing either loop-based variant over our baseline is outlined by the plots in Figure 4.

Instead of a simple geometry shader loop, we also consider methods based on fixed geometry shader instancing. This type of geometry amplification can be achieved by defining a fixed number of geometry shader invocations, which is part of OpenGL's core functionality since version 4.0. While previous literature does not mention any comparable variants, we found this amplification method to work particularly well on NVIDIA GPUs in our use case. Similar to the loop-based pipelines, we have tested combinations with custom frustum and backface culling routines in the geometry shader. On average, we found that the most effective techniques include both of these traits and target layered framebuffers. Based on previous impressions, we also constrained the number of views rendered per invocation down to 4, resulting in an appreciable performance increase. The percentages of cases where these variants outperform our multi-pass baseline are plotted in Figure 5. Of all MVR variants that do not rely on hardware acceleration, $\mathcal{P}(\lceil \frac{N}{4} \rceil, ⊢⇶, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ performed best across all tested NVIDIA GPU models.

4.1.3. OVR and Hardware-Acceleration

The OVR extension allows defining multiple target views for which vertex shader outputs can be written. On NVIDIA Pascal and Turing architectures, choosing 2 or 4 target views respectively allows the extension to exploit the underlying MVR hardware features for maximal efficiency. If more views are defined than the hardware supports, a slower fallback mechanism will be triggered instead. Hence, restricting the number of simultaneous views may again be beneficial to performance in this particular geometry amplification

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

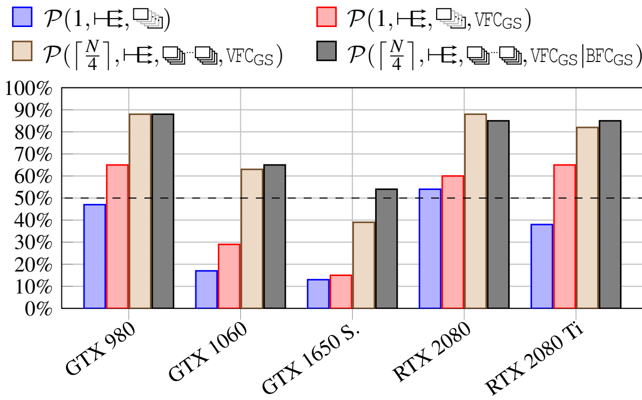


Figure 5: *Percentage of cases where a geometry shader-instancing variant showed better average performance than* $\mathcal{P}(N, \rightarrow, \square\square)$.

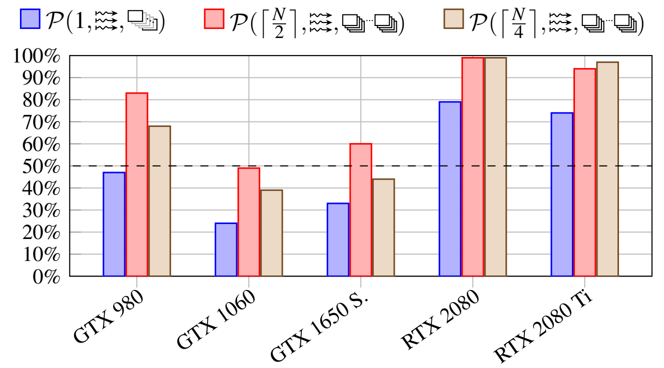


Figure 6: *Bars represent the percentage of configurations where a OVR variant showed better performance than* $\mathcal{P}(N, \rightarrow, \square\square)$.

mode. For instance, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$ describes a variant of MVR that partitions the N views into chunks of four views each and is thus accelerated on Turing. OVR may only be used with array textures and may not be combined with tessellation or geometry shading, which fairly limits the amount of OVR-based MVR pipeline variants. The relative speedup of OVR-based methods over our multi-pass baseline is significant on all NVIDIA GPUs, including models that do not offer MVR hardware support. On older microarchitectures, $\mathcal{P}(\lceil \frac{N}{2} \rceil, \ldots)$ performs better in at least 50% of our tests. Turing GPUs can exploit acceleration for $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$, which has a bigger impact in our assessment since most of our configurations render >4 views. On recent models, both hardware-accelerated variants clearly outperform our baseline in >90% of all test cases. This is confirmed by the information plotted in Figure 6.

4.2. Exploring MVR Setup Parameters and Analysis

We provide a detailed analysis of how setup parameters affect performance of the most promising MVR methods, based on our initial ID buffer evaluation: $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots, \mathrm{VFC_{GS}|BFC_{GS}})$, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots, \mathrm{VFC_{GS}|BFC_{GS}})$ and OVR-based methods. Tables 2 and 4 list the run time results for selected, representative setups. For the full list of timings, please refer to our supplemental material.

4.2.1. Impact of Scene Size

In the results from the NVIDIA models, we could observe that some pipelines are better suited to particular scenes than to others. For the larger scenes ("Bistro", "San Miguel", and "Viking Village"), geometry shader-based pipelines outperform other pipeline variants on Maxwell and Pascal in all test cases and stay within a 20% performance margin to OVR-based techniques on Turing in most cases. $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots, \mathrm{VFC_{GS}|BFC_{GS}})$ turns out to be one of the top-performing techniques in all configurations for the ID buffer tests. Only on high-end Turing GPUs, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$ is often showing significantly better performance. (compare with Table 2). For the remaining, smaller scenes, performance generally varies less across different pipelines. The overall picture shows relative performance gains in favour of OVR-based techniques and our baseline $\mathcal{P}(N, \rightarrow, \square\square)$, compared to geometry shader-based techniques. OVR-based techniques exhibit good performance across all test scenes, but geometry shader-based techniques stay within a performance margin of 20% even on the 2080 Ti in the majority of test cases. While $\mathcal{P}(N, \rightarrow, \square\square)$ falls behind OVR-based techniques in most cases on NVIDIA GPUs, it is the fastest technique on the AMD RX 580 in 100% of tests for the small and large scenes alike.

4.2.2. Differences Across GPU Architecture

Geometry shader-based pipelines show consistent performance characteristics across different NVIDIA GPU microarchitectures. A GPU's performance tier has more impact on the render times than the microarchitecture. This effect is less pronounced with techniques that partition rendering into sets of 4 views ($\lceil \frac{N}{4} \rceil$) and becomes obvious for geometry shader-based "all-in-one" techniques of the type $\mathcal{P}(1, \ldots)$. On low-tier models (i.e. GTX 1060, and GTX 1650 SUPER), they performed worse than $\mathcal{P}(N, \rightarrow, \square\square)$ in 73% of all tests. The optimized pipeline variants of type $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$ show better performance across all NVIDIA GPUs, surpassing $\mathcal{P}(N, \rightarrow, \square\square)$ in virtually all test cases, as can be seen in Table 2.

Since NVIDIA supports the hardware-accelerated creation of up to four views on Turing, we expected performance gains to reflect the doubled number of simultaneous views compared to the Pascal microarchitecture. Indeed, the number of test cases where an OVR variant outperforms other techniques increases on Turing. However, the effect is most noticeable on high-tier NVIDIA GPUs. On the GTX 1650 SUPER, we cannot report an overall preference for OVR-based techniques, since $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots, \mathrm{VFC_{GS}|BFC_{GS}})$ shows better performance in 43% of test cases and roughly equal performance in the others. These relations change drastically with the high-tier models: $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$ performs better in one third of all test cases by a large margin (>20%). Comparing $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$ to $\mathcal{P}(\lceil \frac{N}{2} \rceil, \ldots)$, the former showed advantages on the RTX 2080 and RTX 2080 Ti in 49% and 63% of all tests, respectively. On all other GPUs, the differences between those two OVR-based variants are marginal. Aside from the advantageous performance of $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$ on high-tier Turing GPUs, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots, \mathrm{VFC_{GS}|BFC_{GS}})$ seems to be the best choice for other NVIDIA GPUs across a multitude of different configurations.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

Table 2: *Results of ID buffer generation for different resolutions, scenes, and view counts per GPU. Each table cell represents averaged frame times from 102 measurements per configuration in milliseconds. For brevity, we use the following shorthand to reference our MVR techniques: M.Pass =* $\mathcal{P}(N,\hookrightarrow,\square\cdots\square)$*, GSL*$_4$ *=* $\mathcal{P}(\lceil\frac{N}{4}\rceil,\mapsto\!\lessdot,\square\cdots\square,VFC_{GS}|BFC_{GS})$*, GSI*$_4$ *=* $\mathcal{P}(\lceil\frac{N}{4}\rceil,\vdash\!\equiv,\square\cdots\square,VFC_{GS}|BFC_{GS})$ *and OVR*$_X$ *=* $\mathcal{P}(\lceil\frac{N}{X}\rceil,\approx,\square\cdots\square)$*.*

	Scene	#Views	AMD RX580			GTX 980				GTX 1060				RTX 2080			
			M.Pass	GSL$_4$	GSI$_4$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_4$
1920 × 1080	Bistro	2 × 2	**2.24**	4.48	3.65	2.56	1.95	**1.74**	2.50	2.46	2.50	**2.44**	2.69	1.11	0.96	0.89	**0.88**
		4 × 4	**8.97**	17.88	14.58	10.26	7.81	**7.03**	9.96	8.96	7.85	**7.20**	8.90	4.22	3.67	3.46	**3.32**
		8 × 4	**17.84**	35.66	29.01	20.34	15.60	**14.03**	19.77	17.89	15.51	**14.28**	17.65	8.31	7.16	6.65	**6.62**
	San Miguel	2 × 2	**5.34**	9.74	6.95	6.21	3.87	**3.21**	5.27	5.40	3.75	**3.30**	4.83	3.21	2.07	1.87	**1.68**
		4 × 4	**21.66**	38.58	27.25	27.90	15.61	**13.15**	22.03	26.12	15.20	**13.64**	20.14	16.36	7.03	6.25	**6.09**
		8 × 4	**43.17**	76.76	54.03	57.80	31.33	**26.37**	44.20	54.09	30.50	**27.04**	40.81	34.39	14.59	13.13	**12.6**
	Sponza	2 × 2	**0.72**	1.89	1.79	0.78	0.65	0.63	**0.61**	0.87	1.16	1.01	**0.95**	0.48	0.35	0.37	**0.34**
		4 × 4	**2.99**	7.64	7.21	3.21	2.68	2.58	**2.54**	3.03	3.03	**2.97**	3.08	1.83	1.41	1.37	**1.32**
		8 × 4	**6.02**	15.28	14.40	6.57	5.38	5.17	**5.03**	5.97	5.15	5.01	**4.91**	3.45	2.89	2.91	**2.77**
	Viking Village	2 × 2	**2.62**	4.17	3.40	2.94	2.04	**1.78**	2.82	2.57	**2.11**	2.13	2.68	1.75	1.06	**0.88**	0.96
		4 × 4	**10.41**	16.77	13.54	12.47	8.07	**7.14**	11.57	11.04	7.78	**7.24**	10.38	5.97	3.82	**3.48**	3.55
		8 × 4	**20.65**	33.34	27.17	25.24	16.16	**14.33**	22.99	22.75	15.77	**14.63**	20.89	12.06	7.69	**7.17**	7.32

4.2.3. Varying Number of Views

A very consistent observation across our test results is the dominance of OVR-based techniques for stereo rendering on the Turing microarchitecture. This comes as no surprise, since stereo rendering is the declared purpose of NVIDIAs hardware MVR support. No other pipeline variant was able to outperform $\mathcal{P}(\lceil\frac{N}{4}\rceil,\approx,\square\cdots\square)$ on the RTX 2080 and the RTX 2080 Ti in any of our test cases. On the GTX 1650 SUPER and on the GTX 1060, OVR-based techniques still performed very well in all test cases with two views. For four or more views, geometry shader-based pipeline variants start to show competitive performance characteristics. For 16 and 32 views, they stay within a 10% performance margin to OVR-baesd techniques in most test cases, and often outperform them on pre-Turing microarchitectures. Due to the consistent performance characteristics for $\mathcal{P}(\lceil\frac{N}{4}\rceil,\ldots)$ MVR techniques with four or more views, we argue that they scale comparably well with the number of simultaneously rendered viewpoints. On the AMD RX580, $\mathcal{P}(N,\hookrightarrow,\square\cdots\square)$ again yields the best performance for all view counts (see Table 2).

4.2.4. Raising Vertex & Fragment Load

To establish performance trends of applications with higher vertex load—e.g., vertex skinning—we have simulated highly expensive vertex stages by adding a loop that performs 15k fused multiply-add instructions to the shader. Techniques which amplify geometry before the geometry shader stages are impacted more severely by increased vertex load. Amplifying geometry *in* the geometry shader potentially saves up to $N-1$ expensive vertex shader invocations. Comparing $\mathcal{P}(\lceil\frac{N}{4}\rceil,\ldots)$-type pipelines to $\mathcal{P}(1,\ldots)$-types, the former invoke the vertex shader more often, which is reflected in our measurements: While $\mathcal{P}(\lceil\frac{N}{4}\rceil,\ldots)$-types outperform $\mathcal{P}(1,\ldots)$-types in 100% of test cases with light vertex load by a huge margin, the latter show better performance than the former for many tests with high vertex load across NVIDIA GPUs. OVR-based techniques are coping well with high vertex load on Turing: On the GTX 1650 SUPER, they outperform all other techniques in 100% of tests. On the RTX 2080 Ti, OVR-based techniques perform worse than $\mathcal{P}(1,\vdash\!\equiv,\square,VFC_{GS})$ in 50% of test cases, the same percentage where $\mathcal{P}(1,\vdash\!\equiv,\square,VFC_{GS})$ outperforms $\mathcal{P}(\lceil\frac{N}{4}\rceil,\vdash\!\equiv,\square\cdots\square,VFC_{GS}|BFC_{GS})$. On Maxwell and Pascal, the top-performing techniques are $\mathcal{P}(1,\mapsto\!\lessdot,\square,VFC_{GS})$ and $\mathcal{P}(1,\vdash\!\equiv,\square,VFC_{GS})$. For increased fragment load, we use the results obtained from our G-buffer rendering application (see Table 3). Here, geometry shader-based techniques lose performance compared to OVR-based techniques and—especially on low tier GPUs—also to $\mathcal{P}(N,\hookrightarrow,\square\cdots\square)$. While OVR-based techniques also showed good performance characteristics with light fragment load, they show a clear lead with increased fragment load by the means of G-buffer rendering: Their performance is better than the otherwise well-performing geometry shader-based techniques in 40% of test cases on Maxwell and in up to 90% of test cases on Pascal and Turing. However, the advantage is substantial (>20%) only on high-end Turing GPUs.

4.2.5. Influence of Viewpoint Discrepancy

The fundamental difference between our ID buffer/G-buffer tests and our shadow mapping is the scene setup w.r.t. the view frusta. While for ID buffer tests, view frusta have strong coherence, for shadow mapping, the view frusta might not overlap at all (see Figure 2). The resulting performance measurements for shadow mapping (a selection of which is provided in Table 4) draw a highly interesting—and consistent—picture: $\mathcal{P}(N,\hookrightarrow,\square\cdots\square)$ outperforms OVR-based techniques in the majority of test cases on all NVIDIA GPUs. For all non-high-tier Turing GPUs, we can even observe more than 40% faster frame times in at least half of all test cases (varying per GPU). On high-tier Turing, the performance differences are a bit less pronounced but still substantial (>20% faster in half of all test cases on the RTX 2080 Ti) Geometry shader-based techniques are not doing better with shadow mapping tests. We observed similar relations to $\mathcal{P}(N,\hookrightarrow,\square\cdots\square)$ as with OVR-based techniques. While OVR-based techniques generally show slightly better performance than geometry shader-based techniques, they stay within a 20% margin on most GPUs except for the GTX 1060.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

Table 3: *Results of G-buffer generation for different resolutions, scenes, and view counts per GPU. Each table cell represents averaged frame times from 60 measurements per configuration in milliseconds. For brevity, we use the following shorthand to reference our MVR techniques: M.Pass =* $\mathcal{P}(N,\mapsto,\square\text{-}\square)$*, GSL*$_4$ *=* $\mathcal{P}(\lceil \frac{N}{4} \rceil, \mapsto\!\!\lessdot, ⧉\text{-}⧉, VFC_{GS}|BFC_{GS})$*, GSI*$_4$ *=* $\mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\equiv, ⧉\text{-}⧉, VFC_{GS}|BFC_{GS})$ *and OVR*$_X$ *=* $\mathcal{P}(\lceil \frac{N}{X} \rceil, \approx, ⧉\text{-}⧉)$*.*

			AMD RX580			GTX 980				GTX 1060				RTX 2080			
	Scene	#Views	M.Pass	GSL$_4$	GSI$_4$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_4$
1920 × 1080	Bistro	2 × 2	**2.76**	5.07	4.35	2.40	2.22	**2.15**	2.27	**2.54**	3.07	2.88	2.65	1.57	1.39	1.42	**1.33**
		4 × 4	**11.07**	20.31	17.40	9.46	8.99	**8.61**	9.04	8.33	8.96	8.90	**8.22**	5.74	4.95	4.93	**4.76**
		8 × 4	**22.06**	40.44	34.73	18.92	17.90	**17.20**	18.00	17.03	17.98	17.94	**16.60**	10.80	9.76	9.59	**9.49**
	San Miguel	2 × 2	**6.64**	23.03	15.39	7.43	7.02	**5.82**	6.67	6.59	7.15	6.70	**6.08**	3.97	3.97	3.72	**3.19**
		4 × 4	**26.28**	91.60	61.07	31.06	28.08	**23.23**	26.73	28.50	28.41	26.49	**24.15**	17.67	12.39	11.29	**9.47**
		8 × 4	**52.78**	182.8	121.68	62.36	55.83	**46.11**	53.33	57.48	56.14	52.22	**48.26**	35.97	24.56	22.46	**18.84**
	Sponza	2 × 2	**1.89**	3.32	3.09	1.55	1.50	1.40	**1.35**	**1.73**	1.86	2.49	2.17	1.09	**1.03**	1.08	**1.03**
		4 × 4	**7.61**	13.38	12.44	6.35	5.78	5.70	**5.54**	5.76	5.51	5.48	**5.11**	4.20	**3.70**	3.71	3.72
		8 × 4	**15.22**	26.63	24.77	12.91	11.60	11.27	**10.98**	12.07	11.18	11.11	**10.49**	8.93	7.38	7.42	**7.22**
	Viking Village	2 × 2	**3.82**	10.95	7.84	4.22	3.90	**3.40**	3.94	**3.62**	4.06	3.91	3.73	1.97	1.97	2.01	**1.96**
		4 × 4	**15.08**	43.37	31.40	17.25	15.54	**13.71**	15.61	15.00	16.20	15.72	**14.11**	9.01	7.32	6.95	**6.11**
		8 × 4	**29.98**	86.69	62.71	34.65	31.23	**27.40**	31.13	30.79	32.21	31.35	**28.30**	17.22	14.62	13.63	**12.25**

These performance relations are in stark contrast to the results from ID buffer generation, where $\mathcal{P}(N,\mapsto,\square\text{-}\square)$ shows worse performance than both, OVR-based techniques and geometry shader-based techniques across all NVIDIA GPUs—especially on high-tier models. It appears that NVIDIA GPUs can take advantage of cases where geometry is visible in multiple views that are rendered with the same draw call. With views that do not share geometry, this advantage vanishes, leading to worse performance than $\mathcal{P}(N,\mapsto,\square\text{-}\square)$. A tentative explanation for this phenomenon is given in Section 5.

5. Discussion

While some of the results in Section 4 turned out as expected—e.g., OVR's stereo rendering performance—other results were more surprising: Geometry shader-based pipeline variants of the $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$-type showed very competitive performance for many test cases on NVIDIA GPUs. $\mathcal{P}(N,\mapsto,\square\text{-}\square)$ remains the overall strongest technique for our shadow mapping with incoherent view frusta on NVIDIA models. To provide a deeper understanding of the performance characteristics that we observed for the different MVR approaches, we used the NVIDIA Nsight Graphics profiler on a RTX 2080 and analyzed frame captures for test cases that generate the maximal number of target views in our three applications.

Due to the relatively low load on vertex and fragment shading throughout all tested applications, streaming multi-processor (SM) utilization is never a limiting factor. The $\mathcal{P}(N,\mapsto,\square\text{-}\square)$ approach is limited by the viewport culling (VPC) in all three applications, with raster operations (ROPs) and memory access (VRAM) being high or close to the physical limit. While G-buffer rendering shows the highest ROP utilization, no memory operation reached the hardware limits. The SM utilization caps at 15%, with most load stemming from vertex shading. Geometry shader-based $\mathcal{P}(1,\ldots)$-type pipelines increase SM utilization: it reaches 30% with $\mathcal{P}(1, \mapsto\!\!\lessdot, ⧉, VFC_{GS})$ and 85% with $\mathcal{P}(1, \vdash\!\!\equiv, ⧉, VFC_{GS})$. All other GPU units show low utilization. Most interestingly, VPC is reduced to 15%–30%, which is owed to culling in the geometry shader. In some instances, we found a sudden VRAM overload, although the technique should in theory reduce VRAM access the most. We attribute this effect to temporary scheduling/memory management issues after geometry shading, which may lead to L2 cache thrashing or to overflow of internal work queues. We only observed this issue in some test cases, and only with 32 views being generated. As this issue rarely arises, the general low performance of geometry shader-based $\mathcal{P}(1,\ldots)$-types cannot be attributed to this effect. We believe the main issue is a typical geometry shader problem: if many outputs may be generated, memory management and scheduling become challenging, leaving all profiled units underutilized.

Geometry shader-based $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$-type methods show an overall better balance. Custom culling in the geometry shader reduces VPC pressure significantly compared to our baseline $\mathcal{P}(N,\mapsto,\square\text{-}\square)$. Also, VRAM is in general significantly lower than with our baseline, while it is slightly higher than with $\mathcal{P}(1,\ldots)$-type methods (when it does not spike VRAM). Geometry shader-based $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$-type pipeline variants do not show any sudden VRAM spikes and SM utilization is overall low with 10%-12%. Thus, there is overall no clear bottleneck and it seems that geometry shader output scheduling or simply missing work load may again be the limiting factor for these approaches. In contrast to shadow mapping, the geometry shader output is more coherent in ID buffer and G-buffer test cases, as most often triangles are either culled for all four views or emitted four times. VPC load caps at about 40% to 50%. Shadow mapping on the other hand typically emits one or two triangles, which makes scheduling a lot harder and thus leads to lower performance in this application. Also, VPC pressure remains higher for shadow mapping (up to 75%). However, when able to exploit geometry reuse, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$-type methods find a sweet spot as they reduce the load on the $\mathcal{P}(N,\mapsto,\square\text{-}\square)$ bottleneck and do not lead to too complicated scheduling/memory management issues for the hardware scheduler.

$\mathcal{P}(\lceil \frac{N}{4} \rceil, \approx, ⧉\text{-}⧉)$ shows the highest load on VPC of all tested methods in all test cases (90-100%) All other components see little load, recording the lowest load on everything but SM at 14%-17%.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

Table 4: *Results of LQ shadow mapping for different resolutions, scenes, and view counts per GPU. Each table cell represents averaged frame times from 30 measurements per configuration in milliseconds. For brevity, we use the following shorthand to reference our MVR techniques: M.Pass =* $\mathcal{P}(N, \longmapsto, \square\cdots\square)$, $GSL_4 = \mathcal{P}(\lceil \frac{N}{4} \rceil, \mapsto\!\!\lessdot, ⧉\cdots⧉, VFC_{GS}|BFC_{GS})$, $GSI_4 = \mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\sqsubset, ⧉\cdots⧉, VFC_{GS}|BFC_{GS})$ *and* $OVR_X = \mathcal{P}(\lceil \frac{N}{X} \rceil, \approx, ⧉\cdots⧉)$.

			AMD RX580			GTX 980				GTX 1060				RTX 2080			
	Scene	#Views	M.Pass	GSL_4	GSI_4	M.Pass	GSL_4	GSI_4	OVR_2	M.Pass	GSL_4	GSI_4	OVR_2	M.Pass	GSL_4	GSI_4	OVR_4
800 × 600	Bistro	4	**1.18**	2.15	3.98	**1.37**	2.54	2.44	2.45	**1.35**	3.01	3.02	2.34	**0.50**	0.98	1.11	1.27
		16	**6.00**	11.95	22.47	**6.30**	11.36	11.44	11.60	**5.31**	11.77	11.73	8.96	**2.42**	4.45	5.60	5.32
		32	**12.18**	23.96	43.17	**12.65**	21.54	21.54	21.65	**9.84**	22.05	22.37	16.68	**4.61**	8.42	8.08	10.61
	San Miguel	4	**3.23**	7.83	13.64	**3.39**	5.97	5.85	6.41	**2.93**	5.98	6.08	4.75	4.05	**3.12**	3.47	4.60
		16	**13.39**	34.23	61.54	**14.95**	26.62	26.88	26.58	**13.19**	27.19	27.54	21.73	11.49	**9.66**	11.49	12.29
		32	**25.78**	66.07	118.65	**30.41**	51.61	51.90	51.97	**27.77**	52.95	53.92	40.45	11.49	**9.66**	11.49	12.29
	Sponza	4	**0.46**	0.76	1.12	**0.35**	0.56	0.57	0.62	**0.35**	0.64	0.66	0.52	**0.17**	0.22	0.24	0.24
		16	**1.58**	3.19	4.64	**1.33**	2.23	2.28	2.17	**1.45**	2.75	2.86	2.05	**0.73**	0.95	1.03	1.03
		32	**2.59**	5.70	8.22	**2.62**	4.28	4.43	4.05	**2.63**	5.03	5.12	3.66	**1.43**	2.02	2.32	1.86
	Viking Village	4	**3.44**	7.38	13.00	**3.80**	5.86	5.82	6.22	**3.36**	5.80	5.86	4.92	**1.97**	2.25	2.44	2.81
		16	**7.53**	14.04	25.30	**7.88**	13.42	13.35	14.25	**6.38**	13.91	14.13	10.99	**3.08**	4.67	5.09	5.92
		32	**16.08**	30.91	54.57	**16.82**	28.58	28.35	29.87	**13.16**	29.15	29.17	22.73	**6.40**	10.64	10.44	12.07

In the ID buffer and G-buffer applications, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \approx, ⧉\cdots⧉)$ yields very competitive performance, which we attribute to scheduling also being efficient, as again culling is very consistent and rasterizer queue fill rates are similar. For shadow mapping, the performance is less competitive although the profiling characteristics do not show a vastly different behavior. Our assumption is that scheduling may be an issue, again, as culling clogs the pipeline and triangles trickle into the rasterizer queues of the different views. $\mathcal{P}(N, \longmapsto, \square\cdots\square)$ is also limited by VPC, but generates all load each disjoint view at a time and thus achieves better scheduling and overall utilization.

Our evaluation results indicate that OVR-based techniques show clear advantages with stereo rendering, increased vertex and/or fragment load, and in general on high-tier Turing GPUs. For low vertex and fragment loads, higher numbers of views, and especially on previous NVIDIA microarchitectures, we often found $\mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\sqsubset, ⧉\cdots⧉, VFC_{GS}|BFC_{GS})$ to show slightly superior and consistent performance. It also exhibits relatively good performance with the bigger scenes tested. We recommend this particular pipeline variant in general for use cases that require geometry or tessellation shaders, which are not supported by OVR-based techniques. For non-overlapping view frusta and in general for AMD GPUs, $\mathcal{P}(N, \longmapsto, \square\cdots\square)$ shows the best performance across all tests. We found that overall rendering performance for producing multiple views can drastically be improved by splitting the workload in packages of four views at a time, which is utilized by $\mathcal{P}(\lceil \frac{N}{4} \rceil, \ldots)$-type variants. Although requiring $\lceil \frac{N}{4} \rceil$ times the number of draw calls compared to $\mathcal{P}(1, \ldots)$-type variants, they enable better load distribution and balance as detailed above. Only in combination with very high vertex load or excessive numbers of draw calls, performance shifts in favour of $\mathcal{P}(1, \ldots)$-type pipeline variants. For non-hardware-accelerated MVR techniques, we can state as a general rule of thumb that performing geometry amplification as late as possible is advantageous, and that custom frustum and backface culling in geometry shaders further increases performance. Using layered framebuffers comes with the advantages that clipping can be performed by the rasterizer and early depth tests can be utilized.

6. Conclusion and Future Work

In this paper, we have examined a wide range of different techniques that are available today on modern GPU hardware to achieve multi-view rendering. In order to facilitate the concise description of relevant properties, we have introduced a general and extensible pipeline catalogue. Our evaluation spans multiple GPU generations and use cases, and provides a basis for making informed decisions w.r.t. the applicability of different pipelines, as well as the main factors that impact their performance. In contrast to most available material on this topic, we go beyond stereoscopic projection and analyze multi-view setups that target more than two views. We have shown that, with the help of widely available rendering API features, we can achieve a performance improvement of 5–6× over earlier methods that were explicitly recommended for such scenarios.

While we found NVIDIA's hardware support and the general OVR extension for multi-view rendering to work well across a wide range of setups, we also observed that it can be outperformed in applications with low shading load, particularly on weaker GPU models. Furthermore, the lack of support for tessellation and geometry shading in this feature motivates the question which alternatives can be used if these pipeline stages are required. For those cases, we have identified suitable, optimized methods that are usually within 15–20% of the fastest OVR-based techniques.

For future work, we look forward to extending our catalogue by additional methods that are enabled by the mesh shaders of NVIDIA's Turing architecture. Based on the improvements obtained by the application of custom clipping and culling steps in various pipelines, we are confident that the enhanced programmability of the geometry stage can facilitate further performance gains while preserving the ability to perform custom tessellation and subdivision steps as part of the rendering pipeline. All techniques are available as part of our testbed, which we provide for download and further experiments at https://github.com/cg-tuwien/FastMVR.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

References

[ABC*91] ADELSON S. J., BENTLEY J. B., CHONG I. S., HODGES L. F., WINOGRAD J.: Simultaneous generation of stereoscopic views. In *Computer Graphics Forum* (1991), vol. 10, Wiley Online Library, pp. 3–10. 2

[BS18] BHONDE S., SHANMUGAM M.: Turing multi-view rendering in vrworks. https://devblogs.nvidia.com/turing-multi-view-rendering-vrworks, 2018. [Accessed 19-February-2020]. 1, 5

[BSF10] BECK S., SCHNEIDER M., FRÖHLICH B.: Multiple view generation for auto-stereoscopic displays. In *Proceedings of the 7th Workshop of Bauhaus-Universität Weimar on Virtuelle und Erweiterte Realität der GI-Fachgruppe VR/AR"* (2010), pp. 21–23. 2

[Cas18] CASS EVERITT: Oculus virtual reality multi-view extension. https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview.txt, 2018. [Accessed 6-March-2020]. 1, 2

[Dim07] DIMITROV R.: Cascaded shadow maps. *Developer Documentation, NVIDIA Corporation* (2007). 1

[DNS10] DE SORBIER F., NOZICK V., SAITO H.: Gpu-based multi-view rendering. In *CGAT 2010 - Computer Games, Multimedia and Allied Technology, Proceedings* (2010), pp. 273–279. 1, 2, 4, 6

[Eve16] EVERITT C.: Multiview rendering. In *ACM SIGGRAPH 2016, Moving Mobile Graphics - SIGGRAPH 2016 Course* (2016). 1

[Hal98] HALLE M.: Multiple viewpoint rendering. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), pp. 243–254. 2

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: An efficient multi-view rasterization architecture. In *Proceedings of the 17th Eurographics conference on Rendering Techniques* (2006), Eurographics Association, pp. 61–72. 2

[HSS19a] HLADKY J., SEIDEL H.-P., STEINBERGER M.: The camera offset space: Real-time potentially visible set computations for streaming rendering. *ACM Trans. Graph. 38*, 6 (Nov. 2019). URL: https://doi.org/10.1145/3355089.3356530, doi:10.1145/3355089.3356530. 2

[HSS19b] HLADKY J., SEIDEL H.-P., STEINBERGER M.: Tessellated shading streaming. In *Computer Graphics Forum* (2019), vol. 38, Wiley Online Library, pp. 171–182. 2

[KKSS18] KENZEL M., KERBL B., SCHMALSTIEG D., STEINBERGER M.: A high-performance software graphics pipeline architecture for the gpu. *ACM Trans. Graph. 37*, 4 (Nov. 2018). doi:10.1145/3197517.3201374. 2

[Mar09] MARBACH J.: Gpu acceleration of stereoscopic and multi-view rendering for virtual reality applications. In *Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 2009), VRST '09, Association for Computing Machinery, p. 103–110. URL: https://doi.org/10.1145/1643928.1643953, doi:10.1145/1643928.1643953. 2, 4

[MVD*18] MUELLER J. H., VOGLREITER P., DOKTER M., NEFF T., MAKAR M., STEINBERGER M., SCHMALSTIEG D.: Shading atlas streaming. *ACM Transactions on Graphics (TOG) 37*, 6 (2018), 1–16. 1, 2, 5

[MWH18] MARRS A., WATSON B., HEALEY C.: View-warped multi-view soft shadows for local area lights. *Journal of Computer Graphics Techniques (JCGT) 7*, 3 (July 2018), 1–28. 2

[NVI16] NVIDIA CORPORATION: Nvidia geforce gtx 1080. 3

[NVI18a] NVIDIA CORPORATION: Nvidia turing gpu architecture. 2, 3

[NVI18b] NVIDIA CORPORATION: Vrworks - multi-view rendering (mvr). https://developer.nvidia.com/vrworks/graphics/multiview, 2018. [Accessed 19-February-2020]. 1, 2, 3

[NVI20] NVIDIA CORPORATION: Nsight graphics user guide. https://docs.nvidia.com/nsight-graphics/UserGuide, 2020. [Accessed 6-March-2020]. 5

[RKLC*11] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: Decoupled sampling for graphics pipelines. *ACM Transactions on Graphics (TOG) 30*, 3 (2011), 1–17. 2

[RP94] REGAN M., POSE R.: Priority rendering with a virtual reality address recalculation pipeline. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), pp. 155–162. 2

[SaLY*08] SITTHI-AMORN P., LAWRENCE J., YANG L., SANDER P. V., NEHAB D.: An improved shading cache for modern gpus. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), pp. 95–101. 2

[SS96] SCHAUFLER G., STÜRZLINGER W.: A three dimensional image cache for virtual reality. In *Computer Graphics Forum* (1996), vol. 15, Wiley Online Library, pp. 227–235. 2

[Val03] VALVE CORPORATION: Steam store. https://store.steampowered.com, 2003. [Accessed 26-February-2020]. 1

[Vla15] VLACHOS A.: Advanced vr rendering. In *Game Developers Conference* (2015), vol. 1. 2

[Vla16] VLACHOS A.: Advanced vr rendering performance. In *Game Developers Conference* (2016), vol. 2016. 1

[Wil15] WILSON T.: High performance stereo rendering for vr. In *San Diego Virtual Reality Meetup* (2015), vol. 2015, January 20. 1, 2, 4, 5

[Wil16] WILLEMS S.: Gpu hardware info database. http://gpuinfo.org, 2016. [Accessed 6-March-2020]. 1, 3

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.