Master of Science in Engineering: Game and Software Engineering
June 2018

Exploiting temporal coherence in scene structure for incremental draw call recording in Vulkan

Andreas Flöjt

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Andreas Flöjt
E-mail: anfc10@student.bth.se

University advisors:

Dr. Veronica Sundstedt
Department of Creative Technologies

Stefan Petersson
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. Draw calls in interactive applications are often recorded on a per-frame basis, despite already residing in memory after recording in the previous one. At the same time, scenes tend to be structurally stable; what exists during one frame is likely to exist in the next ones as well.
Objectives. By exploiting the observed temporal coherence in scene structures, this thesis aims to devise a solution to record draw calls incrementally. The purpose of such recording is to reuse what has been recorded previously instead of recording it anew. Two approaches to incremental recording are implemented and compared to regular naïve recording in terms of overhead time. One of them makes use of an extension to the Vulkan graphics application programming interface (API) to evaluate indirect pipeline changes.
Methods. A simulation is used as the method of evaluation, using a simple scene where triangles are rendered in individual draw calls. Two sizes of the scene are used. One matches the upper end of draw call count in samples of modern games and the other is an exaggerated size to test viability for even larger ones. Graphics processing unit (GPU) time is measured along with total execution time to provide numbers on the overhead time caused by the different recording strategies.
Results. When considering the frequency of incremental updates, the multi-draw indirect (MDI) strategy performs very well, outperforming the other strategies even with 100 % updates compared to 0 % of the others. However, it scales poorly with increasing number of pipeline switches, where the other incremental recording strategy performs best instead. In this case, MDI soon becomes more expensive than regular recording.
Conclusions. It is shown that the incremental recording strategies have an observable reduction in overhead time, and may be worth considering. With few pipeline switches, MDI is a viable candidate for its performance and ease of implementation. A large ratio of pipeline switches may not be a realistic scenario, but in those cases the device generated commands (DGC) strategy is a better choice than MDI. Note that the DGC strategy does not perform true incremental recording because calls are still recorded by the GPU. Overhead margins are comparatively low in the smaller scene, but even in that case incremental recording could be beneficial because depending on the implementation, one could avoid traversing parts of data structures that remain unchanged.

Keywords: incremental recording, temporal coherency, overhead reduction, explicit API

Sammanfattning

Bakgrund. Ritanrop i interaktiva applikationer spelas ofta in för varje bild, trots att de redan existerar i minnet från den föregående. Samtidigt har scener en tendens att vara strukturellt stabila; det som existerar för en bild kommer sannolikt existera även för de påföljande.
Syfte. Genom att utnyttja observerad tidskoherens i scenstrukturer avser denna uppsats att utforma en lösning för att stegvis spela in ritanrop. Syftet med sådan inspelning är att återanvända det som spelats in tidigare istället för att spela in det på nytt. Två lösningar för inkrementell inspelning implementeras och jämförs med vanlig naiv inspelning med avseende på kringkostnader. En av dem drar nytta av en utökning till grafikgränssnittet Vulkan för att utvärdera indirekta byten av pipeline.
Metod. En simulering används som utvärderingsmetod, med en simpel scen där trianglar renderas med individuella ritanrop. Två scenstorlekar används. En av dem matchar övre gränsen för antalet ritanrop i ett urval av moderna spel, och den andra är en överdriven storlek för att testa lämpligheten i ännu större scener. GPU-tid mäts tillsammans med total exekveringstid för att tillhandahålla siffror på den kringkostnad som orsakas av de olika inspelningsstrategierna.
Resultat. När frekvensen av inkrementella uppdateringar beaktas presterar multi-draw indirect (MDI) mycket bra, som vinner över de övriga strategierna även med 100 % uppdateringar jämfört med de övrigas 0 %. Den skalar dock undermåligt då antal pipeline-byten ökar, där den andra inkrementella inspelningsstrategin istället presterar bäst. I detta fall blir MDI snabbt dyrare än vanlig inspelning.
Slutsatser. Det visas att de inkrementella inspelningsstrategierna märkbart reducerar kringkostnad och att de därmed är värda att ha i åtanke. Vid få pipeline-byten är MDI en lämplig kandidat tack vare dess prestanda och enkla implementation. Större förhållanden av pipeline-byten är inte nödvändigtvis ett realistiskt scenario, men i dessa fall är device generated commands (DGC) ett bättre val än MDI. Notera att DGC-strategin inte utför sann inkrementell inspelning eftersom kommandon fortfarande spelas in på GPU:n. Kringkostnadsmarginaler är förhållandevis låga i den mindre scenen, men även där kan inkrementell inspelning vara förmånlig eftersom man då, beroende på implementation, skulle kunna undvika att vandra genom delar av datastrukturer som förblivit oförändrade.

Nyckelord: inkrementell inspelning, tidskoherens, reducering av kringkostnad, explicita API

Contents

List of acronyms

API application programming interface. i, iii, 1–6, 9, 20, 31, 35, 36

CPU central processing unit. ix, 1, 4–6, 10, 13, 15–18, 25, 27, 29, 32–35

DGC device generated commands. i, iii, ix, 13, 15–18, 21, 22, 25–29, 31–37

GPU graphics processing unit. i, iii, ix, 1–4, 7, 10, 13, 15–18, 20–22, 25–27, 29, 31–36

MDI multi-draw indirect. i, iii, ix, 17, 20–22, 26–28, 31–37

List of figures

List of tables

Chapter 1

Introduction

In high-performance computing, there is a ubiquitous desire to carry out existing tasks faster and more efficiently. Reducing the amount of work required for some task while achieving the same result frees valuable resources for another. This is particularly important for real-time interactive systems where resources are sparse. It is not sustainable to consume large amounts of execution time because it limits what can be done in a given timespan.

For real-time rendering one often makes use of a widely supported graphics application programming interface (API), the idea of which is to abstract details about the specific hardware on the user's system. Drivers implement the API to communicate with the graphics processing unit (GPU), which ultimately performs the rendering or computation tasks. By programming to the API, programmers can uniformly develop applications that run on a wide range of hardware.

Overhead is present in graphics drivers in the sense that not all computations do useful work. Drivers may have to translate instructions, verify integrity, or generate data, all of which are just means to an end. Such tasks take place on the central processing unit (CPU) and are important in the process of feeding the GPU with work. For draw calls[1], the perceived overhead is not as much actual overhead as it is faster scaling of GPUs than CPUs and thus an inability of the CPU to keep up with the GPU [Wlo03; JL12]. These results are interesting because if draw calls were to be submitted faster, one could either increase performance if CPU bound, or do more useful work in the meantime if GPU bound.

When it comes to the draw calls themselves, one can observe temporal coherence in real-time interactive applications; it would be difficult to interact with a volatile scene. It is therefore expected that geometry rendered in one frame is likely to exist in the next frame as well, albeit slightly repositioned or manipulated, but the same geometry nonetheless. Some structural changes do happen – there may be several of them, but they are comparatively small when put into perspective from the scene in its entirety. This thesis aims to acknowledge the frame-to-frame coherence in order to reduce redundant draw calls by means of incremental draw call recording. The goal is that by submitting draw calls more efficiently, the CPU resources can be routed to other important tasks, such as game logic, physics simulation, or processing artificial intelligence.

[1] When talking about draw calls in this thesis they refer to those calls that represent an intention to render geometry. There are other kinds of calls such as setting state needed for rendering, but the author considers them general calls as opposed to the more specific term *draw call*.

1.1 Background

This section aims to provide an overview of the practical approaches to real-time rendering that has been commonplace for a long time and how they have changed into more explicit ones in recent years.

1.1.1 Interactive rendering

Synthesizing digital images that are interactive or otherwise seem alive is based on the phenomenon that rapidly displaying individual images with slight differences in appearance can provide the illusion of motion, comparable to how an animated story can be made in a notepad by drawing displaced figures and flicking through the pages in quick succession. Using a computer to quickly generate the images based on user interaction is called real-time rendering [AHH08].

Just as for the animated story in the notepad, the same principle applies to real-time rendering. The algorithms that constitute the rendering process generate an image that is displayed on the screen. Generating an image takes time, and the time delta is used to simulate changes in the scene for the succeeding image. It could be the movement of objects during this time or an appropriate change in color as something is heated. Details of what is simulated are of course highly application-dependent, but the point is that the new image is slightly different compared to the previous one. Displaying this sequence of discrete images one after the other is perceived as smooth motion if done quickly enough; the images tend to "meld" together. Higher frame rates—where *frame* is the collective name of the process of simulating, rendering, and displaying a single image—yield even smoother motion, up to about 72 frames per second, at which point individual images tend to be indistinguishable [AHH08].

Achieving 60 frames per second puts large demands on the application, as you are restricted to a time budget of roughly 16.7 ms to simulate, render, and present an image. Modern games boast large quantities of objects to render, potentially consisting of a large number of triangles. However, coping with large data sets is not all there is to synthesizing the images – rendering involves many other tasks such as calculating light interactions [CT82], generating geometry on-the-fly [Bra16], animating objects (as opposed to simply moving or orienting them without deformation, which is much simpler) [JT05], and processing the generated image before presenting it to the screen [RSSF02].

1.1.2 Prior real-time rendering APIs

Implicit graphics APIs such as *Direct3D* and *OpenGL* have for many years been the de-facto APIs for programming real-time graphics. For a long time, graphics APIs exposed what we call *fixed-function pipelines* of hardware at the time [KH13]. In such pipelines, the process of rendering geometry is mostly defined by the hardware itself.

Hardware and APIs later evolved to enable programmability with the introduction of *shaders*; small programs executed on the GPU during a certain stage of the pipeline. APIs have subsequently been expanded to support geometry-, fragment-,

tessellation-, and compute shaders, each working with specific tasks[2]. Fixed-function hardware is still present where programmability makes little sense, like the rasterization process, but where applicable, the trend has been to expand the programmability.

When using these implicit APIs, the programmer generally makes calls to activate the state necessary for rendering—such as what depth tests to use, rasterization configuration, shaders and their resources, and much more—before finally issuing a call to render geometry with the currently active state. While being low-level in comparison to frameworks and engines, the drivers still did much of the heavy lifting behind the scenes, including for example submitting graphics commands to the GPU and synchronizing resource usage.

Drivers are not aware of the specific applications they serve. For that reason, they have to be conservative in nature to work with the limited information available. Because the implicit APIs are stateful—that is, bound state is carried on for upcoming work and changed as needed—the drivers have to continuously verify calls and adapt accordingly due to the lack of up-front knowledge. This causes pure overhead because the computations do no useful work.

Furthermore, the draw calls themselves are issued in isolation, but that does not mean they are actually sent to the GPU immediately; drivers are free to batch calls together and submit them at a time they deem appropriate [Hec15]. The programmer thus lacks control and the drivers have to apply heuristics to determine when is a good time to submit the draw calls. This of course results in sub-optimal execution.

1.1.3 Vulkan and other explicit APIs

Recent explicit graphics APIs like *Direct3D 12* and *Vulkan* were born from demands of the industry and attempts to resolve the issues of prior APIs by delegating control and responsibility directly to the programmer. For the rest of this thesis, Vulkan terminology will be used.

One of the fundamental ideas is that the programmer possesses detailed application knowledge and is for that reason better equipped to determine when draw calls should be submitted to the GPU than the driver, which can only guess. This idea applies to other areas as well – synchronization for instance. The application programmer knows the exact usage patterns of resources and consequently how and when they need to be synchronized. Being granted this explicit responsibility allows the programmer to defer synchronization until it is really needed, allowing intermittent work to execute until then without stalling the pipeline. The programmer can also choose not to unnecessarily synchronize if it is safe to do so.

Draw calls are written into a buffer by the programmer, a process called *recording*. When the programmer decides that the time is right, all commands of the command buffer are explicitly submitted to the GPU for processing. An interesting feature is that by letting the programmer manage the command buffers, one can resubmit previously recorded commands without having to write them again. However, there are some issues in practice, which will be described in Section 1.2.

[2]Among the shaders mentioned, the compute shader is not as specialized as the others, because it is intended to be used for various types of general processing, an area called *general purpose programming on graphics processing units* (GPGPU).

Explicit APIs attempt to reflect how modern hardware works. For example, what used to be a state machine of various rendering states have now coalesced into a single unit (pipeline). This entire unit is created before it is used, and may not be partly changed after being bound; all of it has to be changed at once, removing the need to verify compatibility at runtime because all information has been provided beforehand. It is the programmer's responsibility to acknowledge changes in state by manually organizing pipelines, following a trend in these APIs to provide as much up-front knowledge as possible. This is what makes the APIs explicit; most tasks that are to be performed have to be spelled out. Similar reasoning permeates the concept called *render pass*, which is a means to provide information to the drivers about what kind of resources will be used for rendering and how they are used in subpasses (steps in the application rendering algorithm), but not the exact resources. Such an approach allows drivers to properly prepare their execution because they know what to expect in the future.

Another area of which the programmer is given responsibility is memory management. You are responsible for allocating memory and using it in a safe way. Previously you would tell the drivers what buffers you needed and the allocation in video memory would be done for you, given some hints about intended usage. The programmer is also given control of how and when memory transfers happen. This is useful for techniques such as those presented in this thesis, which carry out detailed memory management.

1.1.4 Indirect rendering

Recent versions of the major graphics APIs (although not restricted to the explicit ones) have supported what is called indirect draw calls. Whereas direct draw calls use the API call itself to pass parameters such as the number of vertices to use when constructing primitives and where to start reading the data from an associated vertex buffer, indirect ones replace those with a GPU buffer handle. It is this buffer that contains the actual parameters to use for the call.

The major benefit of such calls is that by reading parameters from a GPU-resident buffer, draw calls can be specified where the details of what parameters to use are decided by GPU algorithms, while at the same time avoid having to read them back on the CPU. This avoids a costly synchronization between the CPU and GPU. Section 2.2 gives examples of what kinds of algorithms and techniques benefit from indirect rendering.

1.2 Thesis topic

One thing that is still prevalent in graphics backends is the recording of redundant draw calls [HSMT15]. Game scenes tend to change often but little, yet each draw call has to be recorded during every frame. Such an approach to rendering fails to exploit temporal coherence of the scene contents and results in performing work that has already been done before, without any additional gain.

At a first glance, reusing recorded command buffers seems like a viable approach, but reusing them only works as long as they do not change. When a change happens

(for example removing a draw call due to an object no longer being visible), the entire command buffer has to be recorded anew because the programmer may not alter individual parts of an already recorded command buffer.

To alleviate the impact of recording the command buffer from scratch, Vulkan was designed to allow multi-threaded rendering in the sense that recording can be distributed across multiple threads such that each thread records its own secondary command buffer(s) containing a subset of the commands. This only throws more computational power on the problem but does not solve the underlying issue, because changes still require affected secondary command buffers to be recorded again. Ideally, we would like to simply reuse what has already been recorded and only alter what has changed.

That is what this thesis attempts to do. Instead of recording every geometry draw call each frame, it tries to find a solution to reuse what has been recorded, and incrementally record what has changed. A project that was part of a course in 3D-programming explored whether arbitrary, varying geometry could be rendered with a single call in a statically recorded command buffer by representing all geometry with a multi-draw indirect call, whose draw call buffer was maintained separately on the CPU [FNV17]. The results confirmed the hypothesis.

The ability to use a static command buffer with such geometry sparked the idea that individual draw calls could be manipulated without affecting the rest. This thesis builds upon that work to evaluate the possibility of incrementally recording draw calls. Additionally, the thesis aims to generalize the recording by taking pipeline switches into consideration. Indirectly switching pipelines is not possible with core Vulkan, but modern graphics hardware is perfectly capable to do this, most notably that of *AMD* [Wih16] and *NVIDIA* [Kub16]. The latter exposes the experimental `VK_NVX_device_generated_commands` extension, which will be used to evaluate indirect pipeline switching.

Reducing overhead is always of interest in high-performance computing, and this work is particularly relevant for advanced real-time rendering engines that want to further reduce recording overhead than what explicit APIs already do.

1.2.1 Research questions

As part of evaluating incremental recording, the following research questions aim to be answered:

- How can modern graphics API features be used to incrementally record draw calls and decouple them from command buffers?
- How does altering a subset of draw calls perform in contrast to recording the entire command buffer and how does it scale to the worst case of replacing all draw calls?
- How can pipeline switches be taken into consideration in an incremental recording environment?
- How does pipeline switches affect performance in incremental recording?

1.2.2 Ethics and sustainability

By investigating alternative approaches to managing draw calls in visual, interactive applications, this thesis aims to contribute to the field of digital entertainment. What is not theoretical consists of algorithms, and no humans are required for trials. As such, no ethical concerns are expected to be present for this work. The same reasoning applies to sustainability principles because the work is concerned with virtual worlds and makes no claim on physical resources.

1.3 Delimitations

The purpose of this thesis is to provide a proof-of-concept evaluation of the viability in using incremental recording to reduce CPU overhead. Because of this, no effort is done to investigate how to best implement the strategies or how to generalize them. That is, this work does not focus on how to appropriately maintain buffers and render cache, how to integrate with existing engine systems, or supporting structural changes such as added and removed objects. The interesting issue is the performance given a change, not how it is realized.

Part of the design of explicit graphics APIs such as Vulkan includes multithreading support. In Vulkan this is done by allowing multiple threads to record their own secondary command buffers that are later included for execution in a primary command buffer on the main rendering thread. It can be used to speed up recording by letting multiple threads share the workload, but this thesis takes another approach. Instead of leveraging more threads, it attempts to avoid work to begin with. Because changes in a real scenario are expected to be few, the hypothesis is that threading makes little difference for these techniques in typical cases and has thus been excluded from consideration. However, as far as the author is aware, nothing prevents the programmer from threading their indirect buffer maintenance – she just has to work with the data safely like any other threading tasks.

1.4 Thesis outline

Chapter 2 presents related work within the areas of overhead reduction, indirect rendering, incremental computation, and temporal coherence.

Chapter 3 describes the method used to produce the results of this thesis. It includes what scene is used to evaluate the solutions and what times are measured using this scene. The chapter ends with the validity concerns that have been taken into consideration to produce fair results.

Chapter 4 explains the general architecture of the test application to understand how data flows. This is followed by the ideas that form the basis for the incremental recording strategies that operate on the data and practical details regarding their implementation.

Chapter 5 synthesizes measured performance numbers. The main points of interest are how the incremental recording strategies scale with increasing update frequencies and an increasing number of pipeline switches.

Chapter 6 discusses the implication of the produced results and validity concerns. Unexpected outcomes are highlighted, along with reasoning about possible causes. Patterns in results are identified. The chapter ends with a discussion on implementation details, covering some limitations with the solutions.

Chapter 7 answers the research questions and concludes the key points of this work. Suitable use cases for the solutions are suggested, along with possible directions for future work.

1.5 Contributions

The main contributions of this thesis can be summarized in the following points:

- Repurposing indirect rendering facilities normally used for GPU-driven work generation to decouple concrete draw calls from the command buffer.
- Two proof-of-concept variants of incremental recording to exploit temporal coherence in scene structure, one of which includes indirect pipeline switches.
- Performance evaluation of the suggested incremental recording methods compared to the naïve way of recording draw calls.

Chapter 2

Related work

From reviewing the literature, there seems to be little focus on innovative ways to make use of available graphics APIs, or how they can be developed to open new doors for how we interface with graphics hardware. That explicit graphics APIs were born out of industry demands for lower level hardware access—and experiences from console development, which allows such access due to using specific hardware—is further indication of this. It appears to have been an evolution based on real-life issues faced by engineers, but not so much academia. In a sense, one could say that apart from the goal of decreasing recording overhead in real-time rendering, this work attempts to bridge this gap by providing a new perspective on using some parts of graphics APIs in ways not originally intended.

Most of the existing research aims to enable original features or novel algorithms, with the graphics API simply being a means to an end. However, it is not uncommon that performance is degraded in the process, even for work labeled as interactive or real-time. Ideas about how to better leverage graphics APIs to do the same thing but faster (in terms of API overhead) is mostly found in presentations by notable industry actors, but it could prove useful to see research effort targeting the evolution of the tools we use to realize graphics tasks as well.

2.1 Overhead reduction

When it comes to overhead reduction, a common approach is to sidestep the issue by using fewer calls in the first place. To avoid calls also means avoiding the overhead cost that comes along with it. One way to do this is with hardware instanced draw calls, where one call renders multiple instances of the same geometry, instead of submitting individual draw calls. Such functionality can be used to reduce the number of draw calls required when rendering forest scenes by instancing trees several times [ZBM*17].

Instancing is also suggested by Hillaire as a way to reduce the number of calls submitted to the drivers. He further suggests reducing overhead by removing unnecessary state changes [Hil12].

Judnich and Ling reduce the number of draw calls required for terrain rendering by studying geometrical similarities [JL12]. They observe that the subdivision configurations of a quad-tree node are limited and that many of the geometry possibilities are rotationally symmetric when the node consists of vertices in a fixed-size regular grid. This ends up in a small set of unique geometrical patterns that are used to render arbitrary terrain.

Whereas regular indirect draw calls are the same as their direct counterparts except for the source of their parameters, the multi-draw flavor of those calls issue several calls at once by having more than one set of parameters in the GPU buffer. Replacing multiple draw calls with a single one can be used as a way to reduce driver overhead [ESMF14].

2.2 Indirect rendering

Indirect rendering has been around for some time and has traditionally been used as a mechanism to let the GPU generate its own work [Khr17; Mic18], allowing the programmer to decide *when* to render, but not necessarily the details of *what* to render. Note that indirect calls are not strictly necessary because values can be transferred back to the CPU if needed, but they are used for purposes of avoiding such costly GPU-CPU synchronizations. Works that fall into the category of indirect rendering mainly do so because they issue some kind of execution that only depends on values produced by a GPU algorithm.

Crassin and Green use indirect draw calls to process the correct number of vertices (representing threads) for multiple stages of their GPU algorithm without CPU intervention between them [CG12]. To visualize voxels, Pestana uses the Direct3D *DrawIndexedInstancedIndirect* call to render the voxels as instanced cubes, based on which voxels are visible as determined by a compute shader [Pes14].

In the work of Pestana, the shader checks occupancy of voxels to determine whether geometry should be rendered, but the shader can of course also perform more advanced culling. That was done by Jahrmann and Wimmer, who use a compute shader to cull individual blades of grass based on occlusion, distance, and orientation, followed by rendering the passing blades indirectly [JW17].

Doghramachi and Bucci take another approach with occlusion culling, where they render oriented bounding boxes using a reprojected depth buffer to mark which instances are potentially visible. A compute shader uses the visibility information to render visible meshes indirectly. They make further use of indirect rendering to resolve false negatives by indirectly rendering occluded instances using the new full resolution depth buffer [DB17].

2.3 Incremental computation

Incremental computation can be said to be the process of finding a solution to some problem instance, given a solution to a closely related one, by adjusting output based on changes in input [RR93]. Translated to this thesis, given changes in the set of objects to render, we want to tweak the set of recorded draw call commands to respond to those changes.

Wörister et al. use a dependency graph to incrementally update render caches which are, in their case, connected to scene graph nodes and contain for example necessary rendering calls. The dependency graph is built from the scene graph and mediates changes in the latter to the render cache in order to avoid traversing the scene graph [WSMT13]. It could have been a reasonable backing organization to

feed the incremental recording facilities of this thesis, had it been able to work with structural changes, which are of utmost importance for incremental recording. Unfortunately, this is not the case, as it only considers updates in-place.

Haaser et al. maintain a compiled scene representation that adapts to structural changes. Rendering calls are filtered and ordered by state to optimize an instruction stream incrementally. Code is generated to represent calls at the driver level [HSMT15].

Another example of incremental computation is found in [YL13], where a 3D model is produced from a set of initial images and updated as more images are introduced to the set.

A hardware improvement that dynamically adapts to reuse probability of data used by various pipeline stages that reside in last-level caches of graphics processors is suggested by [GSSC13]. It learns reuse patterns to progressively refine the cache management. Similar refinement of output has also been done to simulate physical deformation, where the output is reused as input for further processing [MF12].

2.4 Temporal coherence

Closely related to incremental computation are techniques that exploit temporal coherence. These acknowledge situations that exhibit some level of persistent relativity between values in time. Scherzer et al. provide a survey of techniques that make use of temporal coherence in real-time rendering, noting that the coherency comes from the same surface points being continually visible [SYM*12]. The surface points naturally originate from the same object, meaning that the objects themselves are temporally coherent. The reader is referred to the survey for details on temporal coherence principles.

Chapter 3

Method

To collect real numbers on the performance implications of incremental recording, the strategies were implemented as the rendering backend of a simple scene and measured along with a regular recording strategy (recording using normal draw calls directly into the command buffer). It is conceivable that interviews with companies about their needs could have provided some answers about the feasibility of incremental recording. However, such a method would at best produce highly speculative answers instead of definitive ones. A simulation, on the other hand, produces absolute numbers on how well the technique works in practice from a performance point of view.

Two computers were available for measuring. The first contained a GTX 980 (Maxwell) graphics card with an Intel Core i7-4790K 4.00 GHz processor and 16 GB of RAM. The other contained a GTX 1080 (Pascal) graphics card with an Intel Xeon E5-1620 v4 3.50 GHz processor and 16 GB of RAM. Graphics drivers were at version 397.31. Hardware was chosen to sample architectures where the device generated commands (DGC) strategy (Section 4.3) is supported, but note that it is also available on the earlier Kepler architecture [SLSS17].

3.1 Scene selection

Recording draw calls is a CPU-side operation and the one to focus on for this thesis. Overhead starts to matter when the number of draw calls increases and the scene should thus consist of a large number of them. For ease of implementation, the geometry is largely the same, but they are all rendered using individual draw calls because we are interested in the cases where that number gets large. In a real scenario, identical geometry can be rendered using instanced draw calls, where a single call "stamps out" multiple individual instances of the geometry in question on the GPU. Since this only works for identical geometry, it only helps in some delimited circumstances. In practice, game scenes have lots of varying geometry so you generally end up with a large number of draw calls anyway. Individual draw calls are used in this thesis to simulate the cases where instancing is not possible, without needing actual differences in geometry for each draw call.

The geometry itself can be very simple. Incremental recording aims to reuse what already exists and only update draw calls in response to changes. Only the way we process draw calls and submit them differs; the objects rendered on the GPU are the same regardless of the strategy used to get them there. For that reason, the objects are comprised of single triangles recorded one by one. No particular runtime

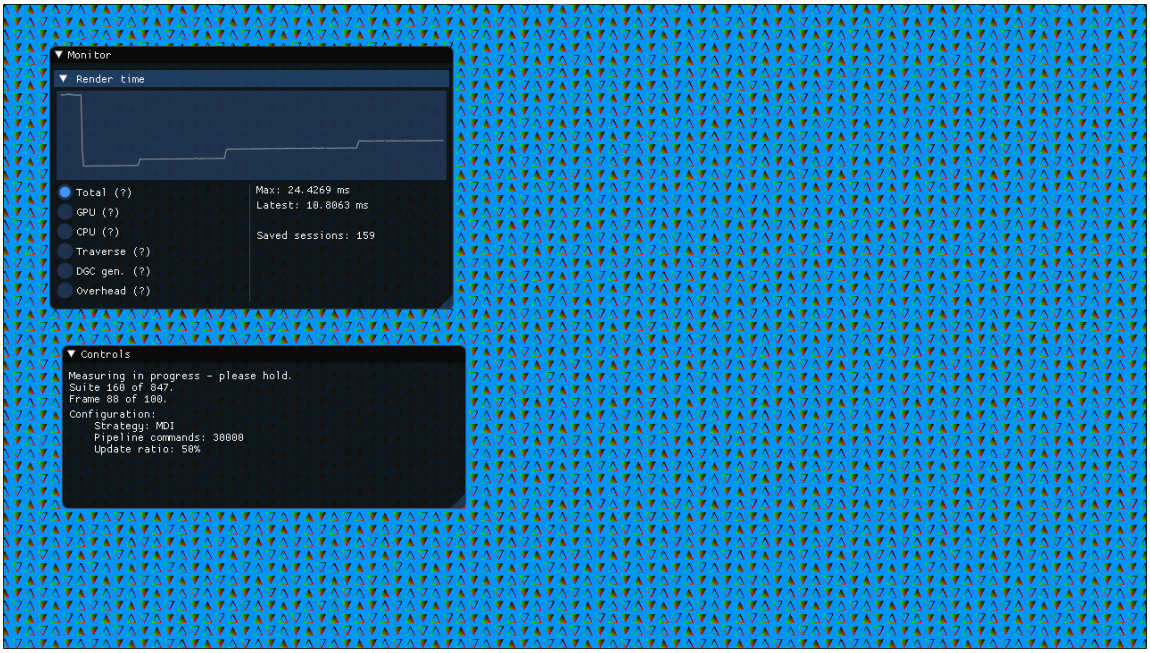


Figure 3.1: The scene used for measuring the recording strategies. In this figure, a measuring session is in progress. This scene consists of 100 000 triangles rendered as individual draw calls.

effects are used, such as lighting or post-processing, but the objects are transformed to position them in a grid. This allows to visually distinguish them from each other.

Changes in geometry are expected to have negligible influence on both recording- and execution time but are included to demonstrate the support of arbitrary geometry in the suggested approaches.

Figure 3.1 shows the scene used for measurements. Every triangle is recorded as an individual unit to stress the draw call count. Some of them are rendered in wireframe whereas others are solid. Every time that changes from one triangle to the next, a new pipeline is bound. This way we can consider the impact of pipeline switches. Issuing a call to bind the same pipeline may be optimized away if the drivers notice that no change is necessary. The distinction between wireframe and solid is therefore used to force a pipeline change when we expect one to happen.

Two sizes of the scene are used. One of them intends to represent a realistic yet demanding scene, based on statistics from recent games (Table 3.1). This scene was chosen to consist of 5 000 draw calls, which outnumbers the average number of draw calls of most of the measured game scenes by a factor $1.35 - 6.61\times$. Due to its strong deviation in draw call count compared to the others, CIVILIZATION was omitted. The other scene size represents an exaggerated case where the scene becomes very detailed and uses 100 000 draw calls to stress recording.

3.2 Measured times

Consider Figure 4.3 in Section 4.2. It shows some of the steps in rendering a frame and illustrates what execution times are collected. Times are measured in milliseconds

Table 3.1: Draw call statistics of some modern games at their maximum graphics settings. The numbers are from the first campaign map of Battlefield 1 and benchmarks of Sid Meier's Civilization VI, Deus Ex: Mankind Divided, The Division, Hitman, and Rise of the Tomb Raider. Courtesy of Stefan Petersson [Pet18].

Game	Max	Average	Std. dev.
Battlefield	942	756	126
Civilization	17 830	17 730	38
Deus Ex	5 859	2 420	703
The Division	8 826	3 709	1 879
Hitman	8 779	1 806	1 388
Tomb Raider	2 123	944	604

and categorized as follows:

- **Total** Complete time from incrementally updating indirect buffers (if applicable) until the image has been rendered.
- **CPU** The part spent on the CPU.
- **Traverse** Traversing the data structure holding the scene objects.
- **GPU** The part spent on the GPU.
- **DGC generation** Command generation time of the DGC strategy.
- **Overhead** The time it takes to record commands, including updating indirect buffers for the incremental strategies and command generation for the DGC strategy.

For the total time, we wait for the device to become idle before ending the measuring. Execution on the GPU is an asynchronous operation to the CPU, meaning that it must be made sure that the GPU has indeed finished before continuing. Just because the end of measuring on the CPU has been reached does not mean that execution has been synchronized with the GPU. This synchronization point happens before presenting the image to the screen because that time is not of interest when measuring overhead. It should be noted that in a real scenario such synchronization should be avoided; the time spent waiting is better used for other tasks.

The CPU time is produced by subtracting the GPU time from the total time. This works as long as execution does not overlap. That is expected to be the case. Memory writes to host visible memory (as is the case for the various buffers written in this implementation) are guaranteed to be visible to the GPU for commands submitted after the write, under the condition that the memory is also host-coherent or that the memory region in question has been explicitly flushed [Khr18].

Traversal time is a pseudo stage. Inside the total time measurement, it is interwoven with the recording time for the indirect strategies; updates happen while the scene objects are iterated. However, the backing structure is irrelevant for the strategies and could be done differently, as suggested in Section 6.5. All strategies

use the same backing storage for uniformity, but some could benefit from a different organization. To get a more fair picture of the *actual* overhead caused by the techniques, the traversal time should be removed from the CPU time. Outside the span of measuring total time, the data structure is therefore iterated once again, this time without doing anything. Although we do not measure the same code that participated in rendering a frame, it should give a good idea of how large a portion of the CPU time can be attributed to traversing the objects.

Queries were used to measure the GPU time. These are hardware counters that can be inserted at various locations in the pipeline. Placing these queries at the beginning and end of the command buffer allows calculating the elapsed GPU time. Originally, the idea was to minimize GPU influence on timings by minimizing shader work and disabling fragment generation, but using queries allows measuring the exact GPU time and take that into consideration.

Using queries also enables more detailed measurements of the impact of the DGC strategy, which is covered in Section 4.3. In short, this strategy has one part where commands are generated by the GPU and another that executes the generated commands. Execution is the same for the other strategies, but generation is different and considered part of recording. Although not strictly required, this process can be split into two distinct steps, and queries allow measuring just the generation time.

Overhead time is the one of interest for this thesis. It is the time a given strategy takes to record and is an indication of how much time is spent doing unproductive work. For the regular strategy, this is simply the CPU time spent recording all draw calls directly into the command buffer. The indirect strategies do not always record as many commands into the command buffer but include additional work considered overhead, such as updating the indirect buffers, transferring buffer memory, and generating commands on the GPU (DGC). It is calculated as

$$overhead = total - gpu - traversal + dgc\ generation.$$

3.3 Validity concerns

When measuring, several configurations of the scene setup and workload are used. These configurations vary in what strategy to use for recording, how many times the pipeline should change during a frame, and how many objects are incrementally updated. Measuring a broad spectrum of configurations allows to identify patterns in execution and observe behavior when scaling the workloads. Single configurations would produce performance numbers, but fail to enable higher level analyses.

All configurations were collected and measured sequentially as part of a suite. When a new configuration was chosen, the first frame was skipped to prevent any potential lingering artifacts from the previous configuration from affecting the measured timings. Execution time is not perfectly stable and varies a bit from frame to frame. For this reason, each configuration was measured and averaged over 100 frames to amortize temporal inconsistencies. Even so, at times one could observe spikes in execution time. The GPU is a single piece of hardware shared by any application that needs to use it. As a step to avoid external factors influencing the measurements, unnecessary processes were terminated and the suite of configurations was measured five times with an application restart between each one. Of the

resulting data set, the smallest and largest values were discarded and the remaining three were averaged to produce a final value.

Modern processors consist of multiple cores that can execute code in parallel, and the internal operating system scheduler may decide to transfer execution from one core to another [SGG14]. To prevent this from happening and potentially result in fluctuating timings, the process was locked to a single core.

To collect as fair measurements as possible, timings are tightly wrapped around the parts we are interested in, with GPU synchronization points in place to make sure measuring intervals start and stop where expected. The timespan covers both recording and rendering, ending before presenting the image to screen. Updates of the scene are implementation details not tied to the techniques of this thesis, and as such, they are excluded from timings. Instead, measuring starts at the time we are ready to record draw calls. Before this happens, a synchronization point makes sure that the graphics device has completely finished its work from the previous frame. When draw calls have been submitted, another synchronization point is used to wait for the GPU to finish its work before ending the measurements. This way we can be sure that the GPU time is included in the total time. Note that such a synchronization point is unnecessary in a shipping product because the GPU can internally synchronize before the image is allowed to be presented, allowing the CPU to continue with other tasks. It is explicitly included in order to collect as detailed timings as possible.

For the DGC strategy, the part that generates the commands is considered overhead. This work happens on the GPU and is measured using its own set of timestamp queries, which had a precision of 1 ns on the available hardware. The measured time is later added when calculating the overhead time. Included in the time range is a barrier used to make sure that writes have finished before the commands are executed.

Strategies attempt to use identical resources for fair measurements. For example, the data structure that stores what objects to render is shared by all strategies. Shaders used when rendering similarly strive to achieve uniformity for the same reason, with the regular and multi-draw indirect (MDI) strategies using the very same one. Unavailability of an extension due to lacking driver support or a bug in their implementation causes the DGC strategy to be implemented slightly differently. This is covered in Section 4.3. The shader itself does not affect the validity of the overhead timings because that happens on the GPU which is measured in isolation and removed from the total time. However, the workaround has a minor impact on the updates of the indirect buffers. Refer to Section 6.2 for a discussion about the implications of the workaround on recording time.

Traversal time for indirect strategies uses a separately measured value from one particular configuration. In the implementation, the code to update indirect buffers includes a branch that answers for a big part of the execution time. The original code to measure traversal time did not account for this branch, meaning that the indirect strategies' overhead time would include implementation details not intrinsic to the strategies themselves. Time constraints prevented an implementation to automatically and properly measure traversal time including the effect of the branch but did allow doing so for one particular configuration. Because traversal is the same for all indirect strategy configurations, this value is expected to be fair and valid for all

of them.

It would have been preferred to keep hardware the same by using only one computer and switch the GPU for the measurements where it was needed. Unfortunately, that was not possible with the resources at hand. This affects comparisons of the DGC strategy overhead on different GPU architectures because differences in CPUs also affect the comparison; not just the GPUs. Relative values between GPU architectures can therefore not be interpreted as the impact of the latter. However, comparing the other strategies' performance should give an idea of how the CPUs affect the measured overhead.

Chapter 4

Implementation

Two indirect strategies were implemented to evaluate the performance of incremental recording when compared to a third regular recording strategy. This section begins by describing the communication flow of the test application to understand where the tasks that are measured happen. This is followed by describing the idea and implementation of the strategies themselves.

4.1 Architecture

A high-level overview of the architecture is presented in Figure 4.1.

`Scene` represents what exists in the world. Any logic that would alter the world in some way—such as deciding where objects are placed, how they respond to user input, or whether they exist at all—belong there. It aims to represent the temporal coherence aspect that this work is based on. Nobody else has access to the state of affairs; the responsibility to act on changes belongs to `Scene`. For this implementation, incremental changes are simulated by marking objects that are active for rendering as "dirty", meaning that they will later be included for incremental recording. Of course, in a real scenario it would be unnecessary to record a call that already exists. Marking objects as dirty was chosen because of two reasons. First, it eases the implementation to focus on measuring the impact of incremental updates without having to deal with new objects or filling holes in the buffers as objects are removed. Secondly, it allows to easily control the exact incremental workload, which is important to observe patterns as the workload increases.

Acting as a middleman between `Renderer` and `Scene`, `RenderCache` represents the set of objects to render. It receives updates to the set from `Scene`. This allows tracking what has changed for when it is time to perform the rendering. All strategies use the same cache, meaning that the underlying data structure stores and traverses objects the same way for all strategies. This is not optimal for some strategies—the indirect strategies, for example, iterate all objects but process only those marked

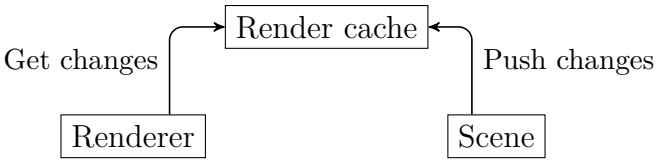


Figure 4.1: Overview of the architecture relevant to the recording tasks. It shows the main actors involved and their interactions with one another.

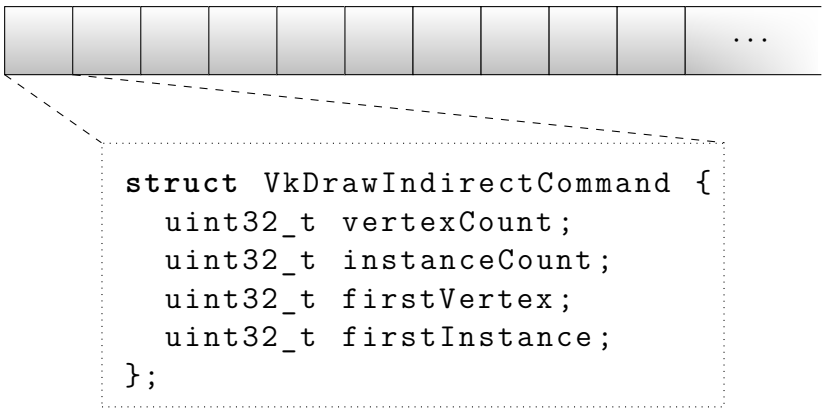


Figure 4.2: Contents of the draw call buffer used for MDI calls. When multi-draw is not available, the same buffer is used, but only the first set of parameters are used.

as modified—but makes traversal overhead consistent and can easily be taken into consideration, as motivated in Section 3.2.

Finally, `Renderer` is responsible for rendering the objects. The incremental recording strategies collect changes from the cache and update their buffers accordingly. This is also where the measuring takes place, as illustrated in Figure 4.3.

`Renderer` manages the various resources used during rendering, one of which is a single vertex buffer. Several meshes live inside the same buffer, and draw calls use a vertex offset to determine which subset of geometry to use. The MDI strategy (Section 4.2) cannot bind vertex buffers indirectly, but a vertex offset can be specified on a per-call basis, allowing the technique to support varying geometry if they reside in the same vertex buffer.

4.2 Incremental recording

The regular recording strategy is used as a baseline for the incremental strategies. It is a naïve approach that records one draw call for each object to render, just as usual. When the pipeline differs, the new one is bound, but if the pipeline has not changed from what was previously bound, nothing happens. The frequency of pipeline switches can be controlled by updating the scene objects accordingly.

The first indirect strategy makes use of MDI capabilities. It is an indirect draw call which works the same as direct ones, with the difference that its parameters are sourced from a GPU buffer instead of passed via the API. Given hardware support, the multi-draw variant works the same, but the indirect buffer now contains draw parameters for multiple draw calls as an array. The memory contents for an MDI-call is illustrated in Figure 4.2.

Sourcing draw call parameters from a GPU buffer is what enables incremental recording because the draw calls have essentially been decoupled from the command buffer. Altering a command buffer that is not recording may not be possible, but buffers used by commands inside is. One just has to remember to properly synchronize resource use, just as any other resources used in Vulkan.

Incrementally recording is simply a matter of exploiting that MDI represents multiple other draw calls and that they are specified in a GPU buffer. Because

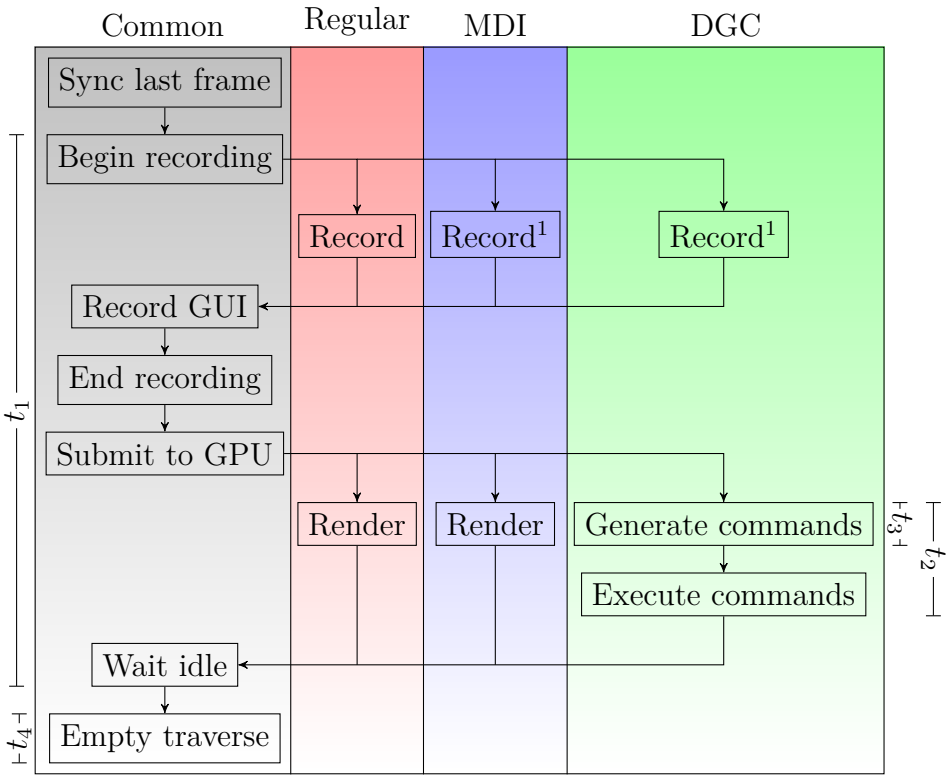


Figure 4.3: Overview of the rendering execution flow. Times t_1, t_2, t_3, and t_4 represent total-, GPU-, DGC generation-, and traversal time respectively.

GPU buffers are persistent, their data remains from one frame to the next, meaning that the draw calls will not have to be written again. Contrary to command buffer contents, the indirect buffer may be manipulated at will. Updating the indirect buffer to be consistent with the changes in the render cache then results in other draw calls being produced by the GPU as the buffer is read by an upcoming MDI call, and we have in effect achieved incremental recording.

The general execution flow can be seen in Figure 4.3. For the regular strategy, the recording stage consists of recording the individual draw calls into the command buffers. The MDI strategy still records into the command buffer, it just coalesces multiple draw calls into one. Before doing this, the indirect buffer is updated according to changes. Other draw calls remain as they are. If a draw call has not changed from the previous frame, it is implicitly reused just by existing in the buffer; it does not have to be recorded again. This is in contrast to regular recording, which *always* records draw calls regardless of whether they were included in the previous frame or not.

4.3 Supporting different pipelines

It was already mentioned that the MDI strategy cannot use different vertex buffers. Draw calls will continue using whatever vertex buffer is currently bound until you

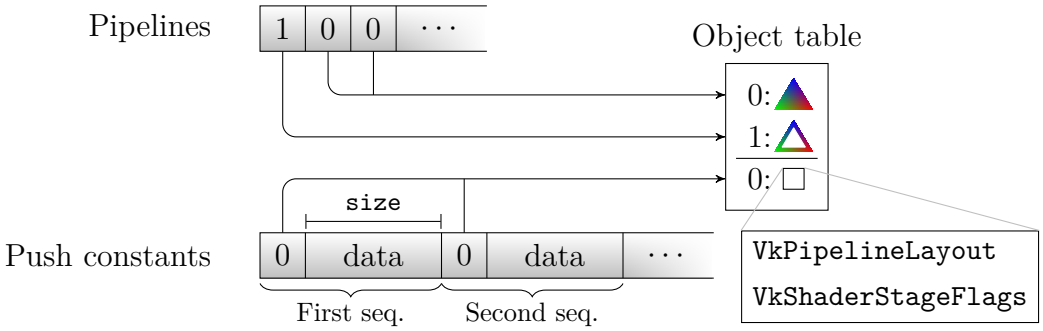


Figure 4.4: Additional indirect buffers for pipeline indices and push constants, used by the DGC strategy. It also uses a draw call buffer, which is the same as for MDI, depicted in Figure 4.2. What subset of push constants to update is specified in the token when defining the sequence layout – the buffer simply provides the actual data.

manually intervene by binding another one. An MDI call results in the same work as if you had issued the draw calls individually, just that the GPU issues the calls by looping over the array of parameters instead. Because MDI only issues draw calls, it has no possibility of binding new vertex buffers, meaning that all of our draw calls have to use the same vertex buffer.

The same problem applies to pipeline switches; MDI simply issues draw calls but has no support of indirectly switching pipelines. How then, does MDI handle different pipelines? The answer is that it does not; we just make more MDI calls as new pipelines are bound. In the worst case where the pipeline is switched for every draw call, you get no benefit of incremental recording at all because an MDI call can only cover one draw before a new one has to be recorded. Add updates of indirect buffers on top of this and you can expect worse performance than regular recording.

A very recent extension called `VK_NVX_device_generated_commands` (DGC), currently in an experimental state, was developed to enable more GPU driven pipelines by allowing it to generate its own work, which includes indirectly switching pipelines [Kub16]. The programmer defines what sequence of commands to generate, and how many times this sequence should be generated. When the time comes to generate the commands, the driver uses an internal compute shader to—given input buffers holding parameters for the commands—write them into a command buffer [KE17].

This extension is the foundation of the third strategy (DGC). The DGC strategy uses the same indirect buffer as MDI for its draw calls, and two additional indirect buffers for pipeline indices and push constants respectively (Figure 4.4). Instead of recording one indirect call for each pipeline, one call is recorded to generate the commands, and another one to execute them. Note that it is not actually required to use two separate calls. You could also execute the commands as they are generated in a single call. Separation of these tasks is done for measuring purposes.

At the time of this work, the `VK_KHR_shader_draw_parameters` extension did not work together with DGC. This extension is used to read the base instance parameter of the draw call and use it as an offset to draw call-specific data. DGC supports generating commands to set push constants, which were used as a workaround to get the same effect. Note that this results in the use of a slightly different shader than

the other strategies, and an additional indirect buffer used for the push constant data that will be fed into the new shader.

Chapter 5

Results

Results are provided for the scene described in Section 3.1. All measurements were done on buffers allocated in host coherent memory. Such memory is available for writing on the host (done during incremental recording) and automatically synchronizes with upcoming commands on the GPU. The implementation supports manually flushing host writes—either complete buffers at once or individual updates—but since all host visible memory was coherent on the tested hardware, manual flushing would be redundant and is not included.

5.1 Invariance in GPU rendering time

The evaluated strategies intend to benefit the CPU side by recording work more efficiently. Work done when rendering is expected to be the same, albeit represented a bit differently, but the same number of final draw calls nonetheless. We have just taken another approach to arrive there. Note that the inability to read the base instance parameter of draw calls with the DGC strategy required it to use a workaround with a different shader. It is therefore expected that differences in GPU rendering time between the strategies should be negligible. Figure 5.1 shows that this assumption was not true for the scene at hand. It shows the rendering time of the large scene for the three strategies as the number of pipeline switches increases from a single call up to one switch per draw call. All strategies have the same number of draw calls and pipeline switches, but they are represented differently in memory. Section 6.1 discusses these results in detail. Nonetheless, the variance in GPU time does not affect other values because it is measured by itself and taken into consideration when calculating the overhead time.

5.2 Scaling with update frequency

One of the main goals of incremental recording is to avoid repeating work by only considering changes in the scene structure. These changes are expected to be few, but could potentially rise in number depending on how volatile the scene is. The best case scenario is when few or no updates at all happen, which means that the scene is structurally stable (perfectly so in the latter case). The worst case happens when very many draw calls have changed and thus need recording. Figure 5.2 shows the overhead time as the update frequency (number of incremental changes) increases. The scene was configured to use a single pipeline command for these metrics. The

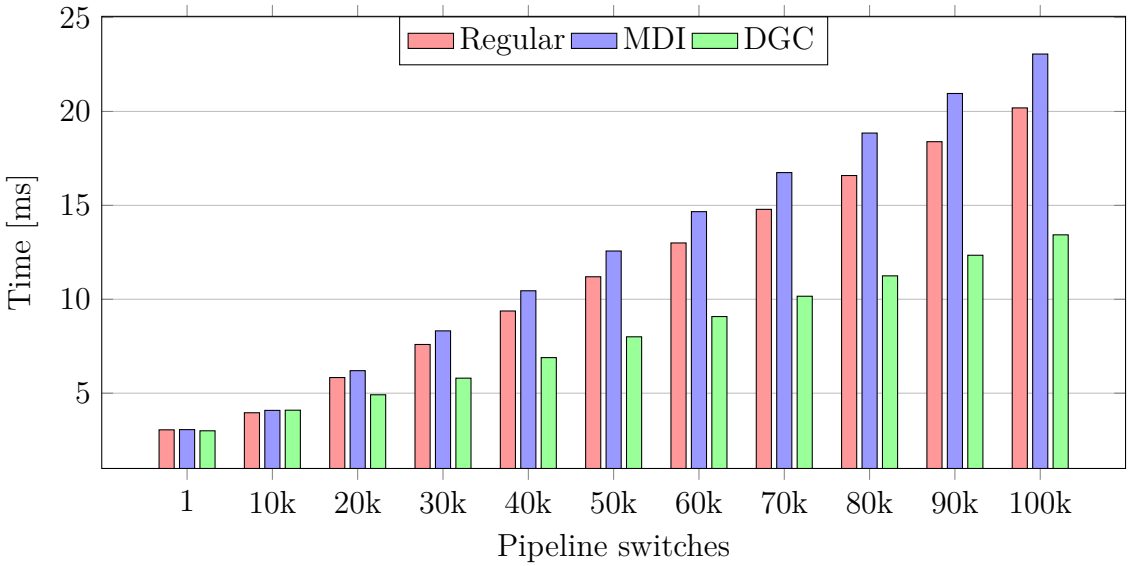


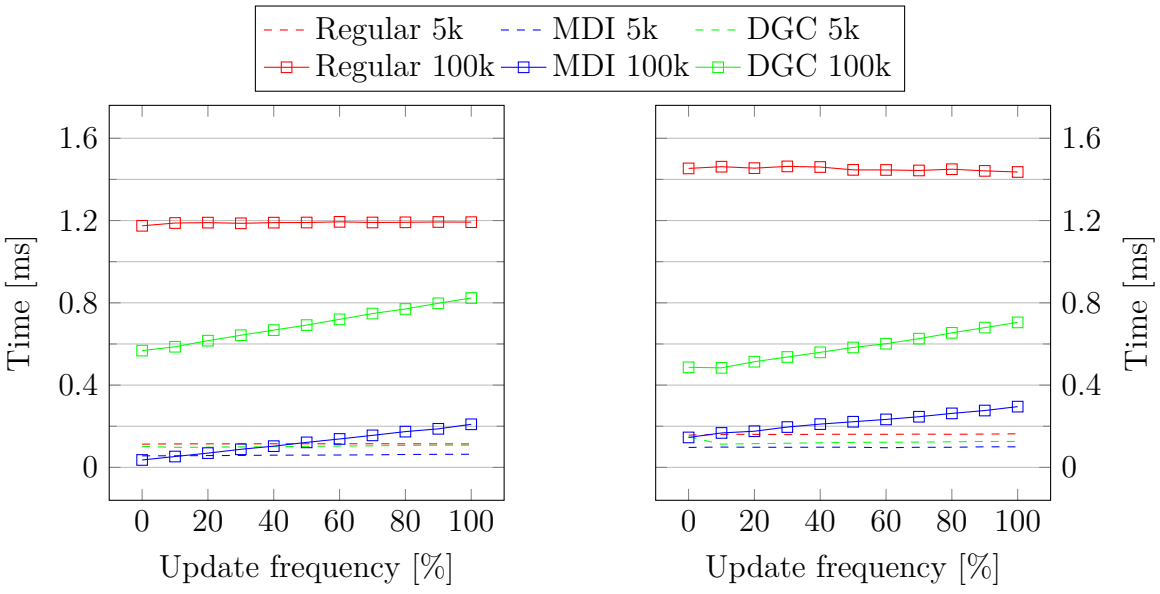
Figure 5.1: How much time the strategies spend rendering on the GPU as the pipeline switch frequency changes. DGC generation time does not apply to rendering and is for that reason not included. This was measured for the large scene on the GTX 980 computer.

left figure shows the overhead time for the three strategies with both scene sizes as measured on the GTX 980 computer. The right figure shows the same but for the GTX 1080 computer. In both figures, the update frequency is the percentage of draw calls that are updated, which for the indirect strategies result in forced recordings and thus a varying number of incremental changes. 0 % updates is the best case as it implies a perfectly stable scene, whereas 100 % updates is the worst case because it means every single draw call has been replaced.

Update frequency variation is implemented such that the first objects of the scene are marked as dirty, where the number of objects to mark corresponds to the update frequency. When it is time to render the scene, dirty objects are forced to result in incremental updates.

5.3 Scaling with pipeline switches

Just as the update frequency of the scene can vary, so can the number of times a new pipeline is bound. These are also expected to be few; in reality we do use several shaders for different kinds of data processing and calculations, but one is generally advised to keep pipeline changes to a minimum. The best case for pipeline changes is when everything can be rendered with only one pipeline – in that case you only bind it once and then record everything without changing it, avoiding the cost of doing so. The worst case happens when every single draw call has to bind its own pipeline because it maximizes the cost of pipeline switches due to no amortization over several draw calls. Overhead time for the strategies as the number of times a pipeline is bound is shown in Figure 5.3. For these metrics, no changes in the scene took place, meaning that no writes happened to the indirect buffers. The left figure



(a) GTX 980 computer (b) GTX 1080 computer

Figure 5.2: Scaling of overhead time as update frequency increases for both scene sizes.

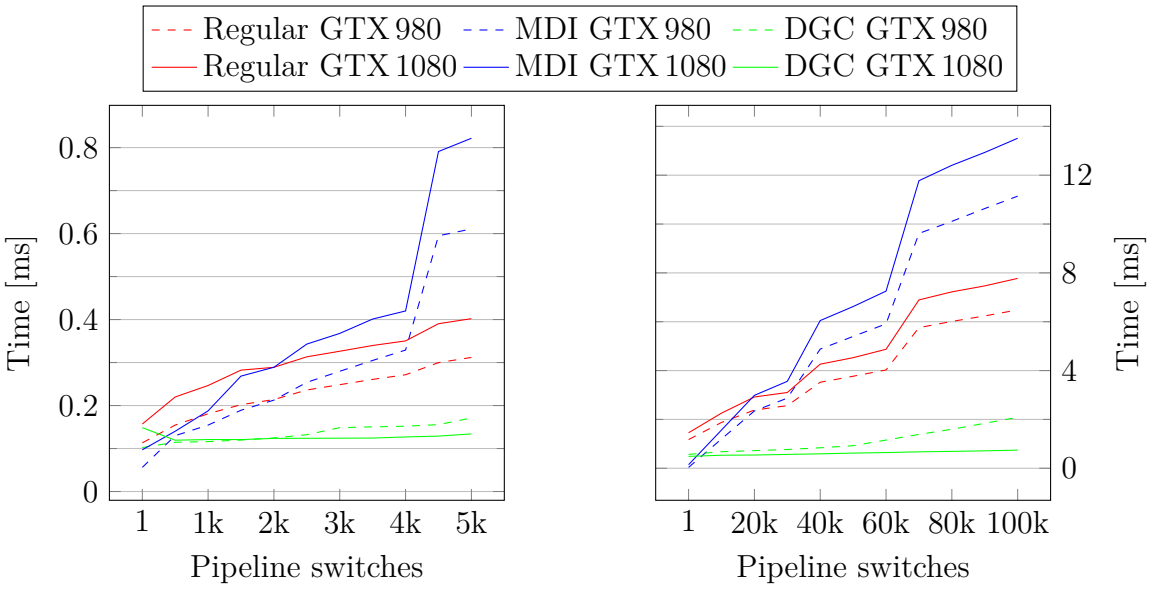
shows all three strategies on both GPU architectures for the small scene and the right figure shows the same but for the large scene. The best case is one pipeline for the entire scene (left part of each figure) and the worst case is one pipeline switch per draw call (right part of each figure).

In practice, when configuring the number of pipeline switches, the objects of the scene are grouped in such a way that when traversed for recording, a new pipeline is bound the desired number of times. Groups have the same size, unless when they do not add up evenly, in which case some groups are one draw call larger.

Varying the update frequency for MDI only affected the management of the indirect draw buffer but not the recording into the command buffer itself. MDI does not support indirectly switching pipelines, meaning that every time another pipeline is used for a set of draw calls, they have to be rendered with a new MDI call. However, this is not the case for DGC, because it does support indirect pipeline switches.

5.4 CPU/GPU distribution for DGC

The overhead for the DGC strategy takes place on both the CPU and the GPU. Whereas the regular and MDI strategies record entirely on the CPU side (differences in how draw calls are processed on the GPU are expected to be negligible), DGC additionally uses the GPU to generate the sequence commands based on the indirect data. This is also part of the overhead. Figure 5.4 shows how the overhead is distributed on the CPU and GPU for this strategy. Each bar represents the complete overhead time for a certain scene size as executed on a certain GPU architecture. They are split into two parts, where the lower part indicates how much is spent on the GPU and the upper part indicates how much is spent on the CPU. They are



(a) Small scene (5k)

(b) Large scene (100k)

Figure 5.3: Scaling of overhead time as the number of commands to bind a pipeline increases, using a stable (non-updating) scene.

repeated for configurations of a single pipeline switch but varying update frequency (top) and no updates but a varying number of pipeline switches (bottom).

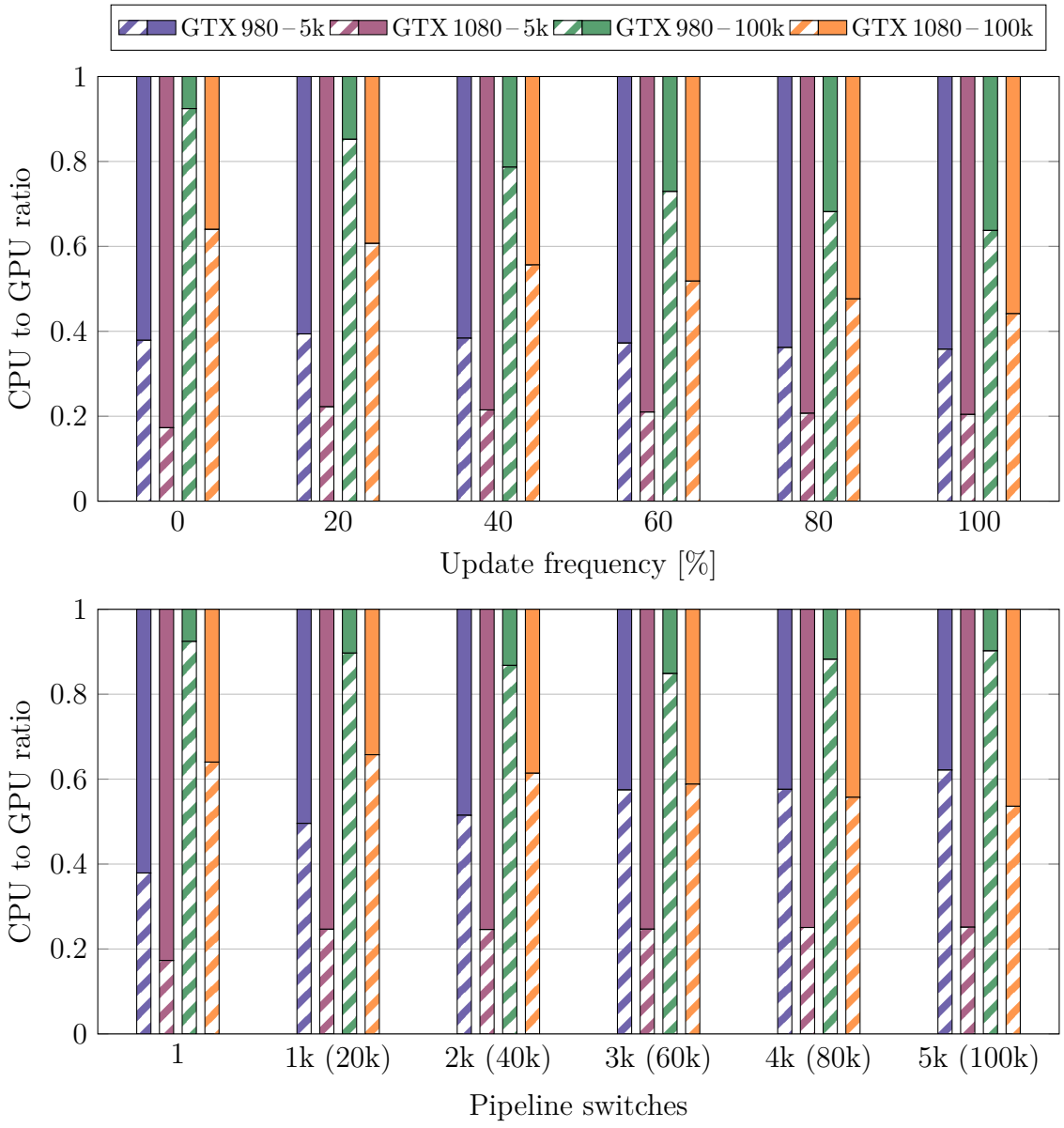


Figure 5.4: DGC overhead distribution on the GPU (hatched) and CPU (solid). For pipeline switches, the numbers in parentheses are used for the large scene.

Chapter 6

Analysis and discussion

In general, one can see that the choice of which strategy to use is not always clear-cut. Different configurations favor one strategy over the other, and implementation details may also shift the scale according to particular needs. What follows are explanations of the results, both in terms of performance and implementation, as well as how they relate to each other.

6.1 GPU rendering time

When synthesizing the measured values, an unexpected discrepancy in the time the strategies spent rendering on the GPU was observed (Figure 5.1). The strategies themselves render the same number of objects using the same pipelines – the difference is in how they record the work to begin with. Two of the strategies, Regular and MDI, use the very same shader. DGC uses a slightly different one for reasons of an unsupported extension which prevents using the same shader as the others. Thoughts on the potential impact of this difference are covered momentarily. Considering that the strategies are largely identical once work has arrived on the GPU, it was expected that the strategies would spend roughly equal time rendering.

While investigating why this could happen, it was found that when increasing the triangle count by rendering sphere meshes of 720 triangles each instead of a single triangle per draw call, the rendering time evened out to show the expected pattern (around 32 ms differing about 1 ms between strategies). A possible explanation of the original measurements is that there could be difficulties in hiding latency that becomes less of an issue as the workload increases. The scene is very simple, with no advanced shading or other computational tasks. For that reason, potential differences in memory access or command processing between the strategies would have difficulties in finding other work to mask the wait. Hiding that latency is easier when more work is available to perform in the meantime, which is exactly what happens with more triangles per draw call. This implies that even if GPU time is irrelevant, one ought to measure it anyway; do not rely on reducing triangle count or disabling fragment generation in the hope that rendering will have a negligible cost. With explicit APIs where one has great control of execution, there is API functionality for fine-grained measurement. Overhead times of this thesis are unaffected by differences in rendering time because the GPU time is measured using hardware counters and taken into consideration when calculating the overhead.

6.2 Impact of DGC workaround

It was mentioned that an unsupported extension required a workaround by using a different shader for the DGC strategy. Instead of reading an object-specific identifier from the base instance parameter, it is passed as a push constant, which is another command DGC may generate. These constants are also passed in an indirect buffer, meaning DGC has more data to upload and is thus at a slight disadvantage. Consider Figure 5.2. DGC increases in overhead time faster than MDI as the update frequency increases. This is attributed to the transfer of more data; pipeline indices and push constants. However, note that in absolute numbers, any additional overhead due to the transfer of more data is negligible, especially in the small scene.

6.3 Overhead due to update frequency

This section covers the results depicted in Figure 5.2. As a reminder, the regular recording strategy simply records everything each frame, because it may not partly alter the command buffer afterward. Updating objects should thus have no effect on the overhead of the regular strategy; they are all recorded no matter what. This can be observed in the figure as horizontal lines, showing the fixed cost of recording a given number of draw calls.

As the update frequency increases, more data is written to the indirect buffers and transferred to the GPU, which is why MDI and DGC scale linearly. This transfer happens very quickly – for the small scene it is not noticeable, and even for the highly exaggerated one, the overhead differs roughly 0.2 ms between no updates and updating every single draw call. Remember that only one pipeline is bound, so MDI records a single call (DGC always does this) for each update ratio, which leads to expect that any differences in overhead between 0 % and 100 % updates are purely due to transferring more data, all else being the same.

The gap between the MDI and DGC strategies is expected because the indirect data written for MDI is a subset of the DGC data. DGC also writes pipelines and push constants in addition to the draw parameters. Furthermore, DGC additionally generates the commands on the GPU and writes them into the command buffer, whereas MDI—to the best of my knowledge—does not.

Comparing the figures, we see that Regular and MDI increase their overhead, whereas DGC decreases it. This is expected; the computers have different processors and the first two strategies operate entirely on the CPU. DGC spends a large part of its overhead on the GPU (Figure 5.4) and therefore benefits from the more recent GPU architecture to result in reduced overall overhead, despite also being negatively affected by CPU differences (Section 3.3).

In a configuration where only update frequency changes, MDI performs the best, even outperforming its non-updating competing incremental recording strategy, despite updating every call itself.

6.4 Overhead due to pipeline switches

Moving focus to pipeline switches in Figure 5.3, the situation is very different from that of the update frequency variation. One still observes the pattern of increased overhead for Regular and MDI, and reduced overhead for DGC when switching from one computer to the other. However, two large observations are made here: pipeline switches generally incur a far greater overhead impact than update frequency, and MDI now becomes the worst strategy for higher frequencies of pipeline switches.

When the number of pipeline switches increases, so does the number of MDI calls. Unlike DGC, this strategy is unable to indirectly set pipelines, and as such, every time the pipeline changes, a new MDI-call has to be issued for the objects using this particular pipeline. In the best case of using a single pipeline, all draw calls are issued with a single MDI-call. With more pipelines bound, fewer draw calls can amortize the MDI-call, meaning the recorded command buffer increasingly resembles that of regular recording, ultimately becoming structurally equivalent at the point where only one draw call is used per pipeline, but with the difference that direct draw calls have been replaced with single-invocation indirect ones. For example, in Figure 5.3a, MDI begins with lower overhead than Regular, eventually breaking even at roughly 2 000 pipeline switches as each MDI-call average only 2.5 draw calls. The performance gap continues to worsen as the scene objects are distributed over a larger number of MDI-calls. Note that for these results, no objects are updated and potential differences in GPU time are removed in the overhead computation, leaving only the CPU time spent recording into the command buffer. Considering that both MDI and Regular record the same number of pipelines, the faster increase of overhead for MDI implies that individual MDI-calls have a higher fixed overhead than direct ones.

DGC scales very well with more pipeline switches, remaining almost constant for the more recent GPU architecture. Although DGC wins for this kind of configuration, it is unlikely to be a realistic setup to use that many different pipelines when rendering the scene. For example, the same lighting model can be used for several materials, and material specifics can often be provided in a data-driven fashion such as textures or shader constants, without requiring a different shader. This translates into fewer pipelines with a higher potential for reuse.

Although MDI performs poorly with pipeline switches, it ought to be possible to make it less of an issue. For example, the deferred+ algorithm decouples shading from geometry processing by replacing material parameters with a material index to perform its specific calculations at a later time [DB17]. With such an approach only a limited set of pipelines that cover vertex shading differences ought to be required to fill the G-buffer because the material that otherwise lives in fragment shaders are taken out of the equation.

6.5 Implementation and limitations

MDI and DGC are similar in the sense that they both manually maintain draw calls in indirect buffers. Changes in draw calls must be sorted to include as many objects as possible in an MDI call. Due to the drivers using a compute shader to generate

commands for DGC, it can more easily handle changes, albeit with unnecessary pipeline switches. However, in practice it is likely a moot point because you are advised to sort draw calls to avoid unnecessary state changes anyway [Kub16].

The manual use of indirect buffers comes at a higher implementation maintenance for DGC. While MDI is fairly straightforward—use one array to store draw call parameters and issue MDI calls that use some subset of it—DGC requires more details to work properly. Pipelines cannot be used directly and must be registered into an object table. This could potentially be troublesome, because it requires knowledge of pipelines up front, or at least an upper limit on the number of them to make the table large enough. One must later make sure that the correct data is found according to the sequence index. In this implementation, the default incrementing sequence indices are used, but administration becomes more troublesome if one uses a user-provided sequence index buffer. Finally, if the generation is separated from execution, one must also reserve memory and generate commands using a secondary command buffer. If indirect pipeline switches are few and fixed, one may want to consider the MDI strategy for its simplicity.

It should be mentioned that while DGC can record commands rapidly, and very much so in the case of pipeline switches, it steals some GPU time for part of its work that would otherwise be done asynchronously on the CPU. It is unclear how MDI compares to this, but the hypothesis about its internal behavior is that a single command is recorded that makes the command processor iterate an array of parameters to execute draw calls on the fly instead of recording draw calls individually based on the source data.

To produce more fair and detailed values for actual overhead, the implementation is not optimized for the respective strategies. Instead, they use a common data structure whose traversal can be measured in isolation to give a better idea of the actual overhead caused by a strategy. For uniform traversal, this was chosen to be a contiguously allocated chunk of memory in the form of a C++ `vector`, which is easy to iterate again without modification to measure the traversal time. All strategies iterate this vector to render objects in turn in whatever way makes sense for them. For the indirect methods, such an approach is not ideal in an optimized implementation because you traverse everything, modified or not. Instead, it would be reasonable to exploit a data structure that allows only iterating changes. Avoiding unnecessary traversal could be beneficial and therefore make the incremental strategies useful even for small scenes, despite recording already being decently fast in Vulkan.

Chapter 7

Conclusions and future work

In this work, two solutions for incrementally recording draw calls in Vulkan—one using MDI calls and another using the DGC extension—were proposed and evaluated in comparison to the naïve way of recording, where draw calls are individually and repeatedly recorded directly into a command buffer. This was done to answer whether incremental recording could be done at all, and if so, whether the recording overhead could be reduced. Evaluation was done by implementing the strategies and measuring their recording overhead for different numbers of updated draw calls and switched pipelines. The incremental strategies were shown to drastically reduce recording overhead under proper circumstances. To the author's knowledge, incremental recording is a novel strategy that has not been published before. This work may be of interest to developers of high-performance, interactive rendering engines with a desire to reduce recording overhead, such as those used in modern games.

The research questions were answered as follows.

Research question 1

How can modern graphics API features be used to incrementally record draw calls and decouple them from command buffers?

Manually writing into GPU buffers from the CPU side allows using indirect calls to decouple draw calls from command buffers. The persistent nature of GPU buffers naturally supports incremental recording by only updating relevant parts, exploiting that others remain unaltered.

Research question 2

How does altering a subset of draw calls perform in contrast to recording the entire command buffer and how does it scale to the worst case of replacing all draw calls?

Incrementally recording draw calls performs very well compared to regular recording, with MDI being the fastest of the two incremental strategies. DGC has a higher cost due to larger memory transfers and an extra step to generate commands on the GPU. In that regard it is not true incremental recording as every draw call is still recorded – it just takes place on the GPU. Both are faster than regular recording even when every draw call is written, and MDI with 100 % updates still outperforms DGC with no updates in terms of overhead time.

Research question 3

How can pipeline switches be taken into consideration in an incremental recording environment?

Similarly to the first research question, this question is also answered by the implementation itself but differs between the two indirect strategies. DGC natively supports pipeline switches by being able to generate commands for binding pipelines. The programmer has to register them in a table, but when that is done the pipeline can be switched indirectly. MDI requires new indirect calls when a pipeline has been switched. This resembles regular recording, but with the difference of representing multiple draw calls with the same indirect one (as long as their parameters are stored contiguously in the GPU buffer).

Research question 4

How does pipeline switches affect performance in incremental recording?

When pipeline switches are introduced with the final question, MDI performance starts to dwindle whereas DGC scales very well. Due to the inability of MDI to indirectly switch pipelines, new MDI calls have to be recorded every time a new pipeline is bound. It is concluded that individual MDI calls have a higher overhead cost than direct draw calls, which means that as the number of draw calls that can amortize this higher cost decreases, the overhead will soon surpass that of regular recording. DGC, on the other hand, only records a single call even with pipeline switches, and the command generation done by the GPU is not affected very much as that number increases. This is even more apparent on the more recent GPU architecture.

Incremental recording shows a noticeable reduction in overhead, particularly in larger scenes. It is expected that a major part of draw calls remain from one frame to the next, and in those cases MDI performs extraordinarily well (the lower frequency parts of Figure 5.2). Surprisingly, it performs well even when updating every draw call. Unfortunately, MDI quickly loses its edge when a large fraction of draw calls require different pipelines. DGC can be used when pipeline variations are needed, but one must keep in mind that it is more difficult to maintain and work with because it uses many objects and buffers that must play well together, requires upfront knowledge of what pipelines to use in order to accommodate them in the object table, and still records everything, but takes some GPU time to do it. MDI, on the other hand, requires just the indirect buffer and a single call of the API.

Although recording commands by normal means is reasonably fast in Vulkan, the incremental methods may still be worthwhile to pursue. Designing around such use could affect memory access patterns to only visit changes instead of everything.

7.1 Future work

Because the main idea of incremental recording is to adapt to changes in the render set, one must consider additions and removals in a real scenario. These may happen in an arbitrary order, which can produce holes in the indirect buffer or result in draw call parameters that are unsorted with respect to pipelines. Parameters must be contiguously stored for MDI, whereas DGC has more leeway because it generates the necessary pipeline commands and allows the programmer to provide what sequence indices to generate. This work covers the performance impact of varying amounts of changes but does not generalize buffer maintenance to support additions and removals. A suggested direction for future work is to investigate practical considerations for how to manage the indirect draw call buffers in the face of structural changes, where splitting the indirect buffer into buckets belonging to a certain pipeline is one option.

Another perspective on buffer management is to keep it sorted as changes happen. Just as for the incremental recording in this work one could exploit temporal coherence when sorting; an indirect buffer that was sorted with respect to pipelines in one frame is likely to remain mostly sorted with few local changes during the next. This is particularly interesting for DGC because it does not have a hard requirement of the data being completely sorted. If some sequences are not grouped properly they will make an unnecessary pipeline switch at runtime, but the rendering result will still be correct. That may be acceptable while amortizing sorting the source data over several frames. However, MDI will still require a fully sorted buffer to avoid using draw calls with the wrong pipeline bound.

Bibliography

[AHH08] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering*. 3rd ed. A K Peters, 2008.

[Bra16] W. Brainerd. "Catmull-Clark Subdivision Surfaces". In: *GPU Pro 7. Advanced Rendering Techniques*. Ed. by W. Engel. A K Peters/CRC Press, Apr. 27, 2016.

[CG12] C. Crassin and S. Green. "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer". In: *OpenGL Insights*. Ed. by P. Cozzi and C. Riccio. CRC Press, July 2012. Chap. 22.

[CT82] R. L. Cook and K. E. Torrance. "A Reflectance Model for Computer Graphics". In: *ACM Transactions on Graphics* 1.1 (Jan. 1982).

[DB17] H. Doghramachi and J.-N. Bucci. "Deferred+". In: *GPU Zen. Advanced Rendering Techniques*. Ed. by W. Engel. Encinitas, CA: Black Cat Publishing Inc., 2017.

[ESMF14] C. Everitt, G. Sellers, J. McDonald, and T. Foley. "Approaching Zero Driver Overhead". GDC Presentation. Game Developers Conference. Mar. 2014. URL: http://gdcvault.com/play/1020791/ (visited on 04/26/2018).

[FNV17] A. Flöjt, J. Nilsson, and P. Valtersson. "Reducing CPU Overhead Using Vulkan Indirect Draw Calls". Project report in 3D-programming course. May 25, 2017.

[GSSC13] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri. "Efficient management of last-level caches in graphics processors for 3D scene rendering workloads". In: *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2013, pp. 395–407.

[Hec15] T. Hector. *Vulkan: Explicit operation and consistent frame times*. Dec. 7, 2015. URL: https://www.imgtec.com/blog/vulkan-explicit-operation-and-consistent-frame-times/ (visited on 06/06/2018).

[Hil12] S. Hillaire. "Improving Performance by Reducing Calls to the Driver". In: *OpenGL Insights*. Ed. by P. Cozzi and C. Riccio. CRC Press, July 2012. Chap. 25.

[HSMT15] G. Haaser, H. Steinlechner, S. Maierhofer, and R. F. Tobler. "An incremental rendering VM". In: *Proceedings of the 7th Conference on High-Performance Graphics*. HPG '15. ACM, 2015.

[JL12] J. Judnich and N. Ling. “Symmetric Cluster Set Level of Detail for Real-Time Terrain Rendering”. In: *2012 IEEE International Conference on Multimedia and Expo*. 2012.

[JT05] D. L. James and C. D. Twigg. “Skinning Mesh Animations”. In: *ACM Transactions on Graphics* 24.3 (July 2005).

[JW17] K. Jahrmann and M. Wimmer. “Responsive Real-time Grass Rendering for General 3D Scenes”. In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’17. San Francisco, California: ACM, 2017.

[KE17] C. Kubisch and I. Esser. “Vulkan Technology Update”. GTC Presentation. GPU Technology Conference. 2017. URL: http://on-demand.gputechconf.com/gtc/2017/presentation/s7191-kubisch-esser-vulkan-technology-update.pdf (visited on 05/04/2018).

[KH13] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Ed. by T. Green. 2nd ed. Elsevier Inc, 2013.

[Khr17] Khronos. *Vertex Rendering*. Khronos OpenGL Wiki. Sept. 2017. URL: https://www.khronos.org/opengl/wiki/Vertex_Rendering#Indirect_rendering (visited on 02/28/2018).

[Khr18] *Vulkan® 1.0.74 – A Specification*. The Khronos Group Inc. Apr. 21, 2018. URL: https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html (visited on 05/06/2018).

[Kub16] C. Kubisch. *Vulkan Device-Generated Commands*. NVIDIA GameWorks blog post. NVIDIA. Nov. 8, 2016. URL: https://developer.nvidia.com/device-generated-commands-vulkan (visited on 05/25/2017).

[MF12] M. M. Movania and L. Feng. “Real-Time Physically Based Deformation Using Transform Feedback”. In: *OpenGL Insights*. Ed. by P. Cozzi and C. Riccio. CRC Press, July 2012. Chap. 17.

[Mic18] Microsoft. *Direct3D 11 Features*. DirectX documentation. 2018. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476342(v=vs.85).aspx (visited on 02/28/2018).

[Pes14] A. Pestana. *Voxel visualization using DrawIndexedInstancedIndirect*. Personal blog. Mar. 18, 2014. URL: http://www.alexandre-pestana.com/voxel-visualization-using-drawindexedinstancedindirect/ (visited on 02/28/2018).

[Pet18] S. Petersson. Personal communication. Apr. 18, 2018.

[RR93] G. Ramalingam and T. Reps. “A Categorized Bibliography on Incremental Computation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’93. Charleston, South Carolina, USA: ACM, 1993, pp. 502–510.

[RSSF02] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. “Photographic Tone Reproduction for Digital Images”. In: *ACM Transactions on Graphics* 21.3 (July 2002).

[SGG14] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Internation Student Version. 9th ed. John Wiley & Sons, 2014.

[SLSS17] M. Schott, T. Lorach, K. Spagnoli, and N. Subtil. "NVIDIA Vulkan Update". GDC Presentation. Game Developers Conference. Mar. 3, 2017. URL: http://www.gdcvault.com/play/1024360/ (visited on 04/24/2018).

[SYM*12] D. Scherzer, L. Yang, O. Mattausch, D. Nehab, P. V. Sander, M. Wimmer, and E. Eisemann. "Temporal Coherence Methods in Real-Time Rendering". In: *Computer Graphics Forum* 31.8 (2012), pp. 2378–2408.

[Wih16] G. Wihlidal. "Optimizing the Graphics Pipeline with Compute". GDC Presentation. Game Developers Conference. Mar. 2016. URL: http://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/29204330/GDC_2016_Compute.pdf (visited on 05/25/2017).

[Wlo03] M. Wloka. ""Batch, Batch, Batch:" What Does It Really Mean?" GDC Presentation. Game Developers Conference. Mar. 2003. URL: http://www.nvidia.com/docs/io/8230/batchbatchbatch.pdf (visited on 04/16/2018).

[WSMT13] M. Wörister, H. Steinlechner, S. Maierhofer, and R. F. Tobler. "Lazy Incremental Computation for Efficient Scene Graph Rendering". In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: ACM, 2013, pp. 53–62.

[YL13] Z.-H. Yuan and T. Lu. "Incremental 3D reconstruction using Bayesian learning". In: *Applied Intelligence* 39.4 (Dec. 1, 2013), pp. 761–771.

[ZBM*17] X. Zhang, G. Bao, W. Meng, M. Jaeger, H. Li, O. Deussen, and B. Chen. "Tree Branch Level of Detail Models for Forest Navigation". In: *Computer Graphics Forum* 36.8 (Feb. 7, 2017).

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden