An Incremental Rendering VM

Georg Haaser[1,2] Harald Steinlechner[1] Stefan Maierhofer[1] Robert F. Tobler[1]

[1]VRVis Research Center* [2]Vienna University of Technology

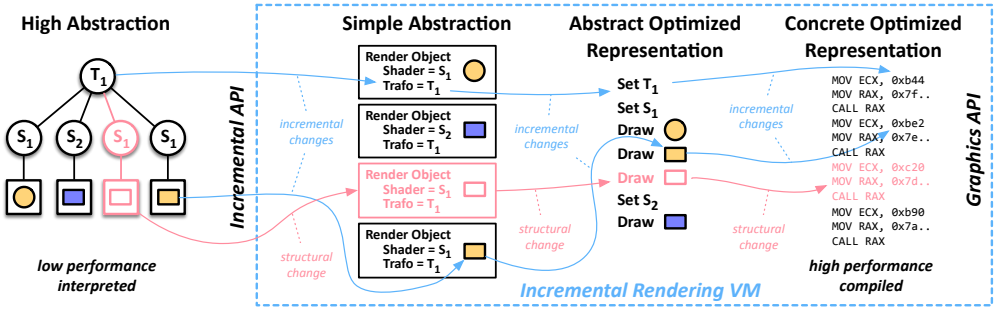


Figure 1: *Successive lowering of abstraction: the rendering intent of the programmer is successively mapped into representations of lower abstraction, starting from a high-level representation (e.g. a scene graph) on the left to the compiled calls to the graphics API on the right. By tracking the incremental changes (modified values) and structural changes (additions and removals) between these different representations, we exploit temporal coherence and avoid repeatedly performing the same optimizations, resulting in a significant speed-up.*

Abstract

We introduce an incremental rendering layer on top of standard graphics APIs such as OpenGL or DirectX in the form a virtual machine (VM), which efficiently maintains an optimized, compiled representation of arbitrary high-level scene representations at all times. This includes incremental processing of structural changes such as additions and removals of scene parts, as well as in-place updates of scene data. Our approach achieves a significant framerate increase for typical workloads and reasonable performance for high-frequency changes. Processing is performed in running time $O(\Delta)$, where Δ is proportional to the *size* of the change and the optimized representation has no runtime overhead with respect to the underlying graphics API. This is achieved by tracking and applying all changes as incremental updates to appropriate data structures and by adaptively synthesizing a program of abstract machine code. In a final step this abstract program is incrementally mapped to executable machine code — comparable to what just-in-time compilers do. Our main contributions are (i) an abstract interface for rendering and visualization systems enabling incremental evaluation, (ii) adaptively optimized abstract machine code in the context of stateless graphics commands, and (iii) subsequent adaptive compilation to executable machine code including on-the-fly defragmentation.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems D.3.4 [Programming Languages]: Processors—Incremental compilers, Optimization

Keywords: rendering, optimization, dynamic compilation, virtual machines

*{haaser|steinlechner|maierhofer|tobler}@vrvis.at

High Performance Graphics 2015, August 7 – 9, 2015, Los Angeles, CA.

ACM 978-1-4503-3707-6/15/08 $15.00
http://dx.doi.org/10.1145/2790060.2790073

1 Introduction

With ever increasing rendering performance of graphics hardware, optimization of rendering and visualization systems becomes more and more important. While GPUs steadily increase their throughput, real-world performance of rendering or visualization systems often cannot keep up due to various implementation overheads. In order to resolve this problem many state of the art engine and driver implementations perform a wide range of optimizations on the fly, such as automatically filtering out ineffective or redundant graphics instructions.

However, applications implemented on top of *high-level* programming interfaces which provide a productive development environment (e.g. scene graphs) still fall short of theoretical peak hardware performance. This leads to the common practice of *hand-tuning* graphics code or of circumventing high-level APIs altogether in order to achieve the desired performance. As a result, programmability and productivity suffers due to the lack of high-level abstraction mechanisms and developers need to manually tune low-level graphics commands for different use cases.

We propose that a significant fraction of the remaining mismatch between soft- and hardware throughput can be eliminated by introducing *a proper notion of incremental change* allowing to maintain temporal coherence of low-level instruction streams.

Graphics APIs such as OpenGL or DirectX operate on *mutable shared state*, i.e. graphics commands alter parts of the rendering state implicitly. In order to know the full rendering state relevant for an optimization, basically all previous commands need to be considered, since any command could have altered parts of the relevant state. This significantly complicates optimizations. Presumably the Khronos Group's Vulkan API [Khronos 2015] takes a significant step towards *stateless graphics* which will simplify such optimizations, but at the time of writing there neither exists published research nor a publicly available API to support such claims.

Irrespective of the actual graphics API, the current notion of a sequence of graphics commands that need to be executed is too simple an abstraction to support optimizations that deal with temporal coherence. To that end we have identified the following fundamental issues with current low-level graphics programming interfaces:

- Backend implementations need to execute redundant calls and simulate their semantics in order to filter out redundant ones. Even if redundant calls can be detected fast in the driver, the rendering engine wastes cycles by actually executing the useless command. As shown by [Wörister et al. 2013], issuing ineffective commands has a significant overhead, especially in managed execution environments such as the JVM, .NET or in browsers executing JavaScript.
- Although graphics APIs provide so-called state objects, not all changeable input is represented in these state objects. Due to dependencies between these different forms of input, optimal code for submission to the API needs to contain dependent control flow, making optimization difficult.
- Finally, graphics backends traditionally execute all draw calls each frame, over and over again. As a result, all optimizations need to be carried out repeatedly, instead of exploiting temporal coherence between successive frames.

In order to automatically perform all relevant optimizations necessary to issue a near-optimal stream of graphics commands to the low-level graphics API, we propose a novel programming interface with a design based on the following premises:

- Graphics commands have no implicit state. Each effective draw call captures the complete rendering state. As a consequence, draw calls can be analyzed independently of their occurrence.
- Command arguments explicitly understand the notion of changes, which is crucial for adaptive optimization.

By introducing a new layer on top of existing graphics APIs we can overcome the fundamental problems of the current architecture, and achieve fully automatic optimization.

Our contributions are:

- An abstract **interface for rendering systems**, which acts as the smallest common module for other high-level front-ends such as scene graphs and captures all necessary functionality while naturally **enabling incremental evaluation** (Section 2).
- The concept of **adaptively optimized abstract machine code** in the context of stateless graphics commands. While abstract machine code for rendering is a well known technique [Wörister et al. 2013], we utilize incremental evaluation for code generation (Section 3.2).
- Based on an **analysis of virtual machine interpreters** (Section 4.2) for executing the graphics commands implied by our new interface, we decided to employ **adaptive compilation** to minimize the overhead of submitting graphics commands (Section 3.3)
- Although initially optimal, the generated code will degrade over time due to additions and removals of graphics commands based on incremental changes. We use **on-the-fly defragmentation** of optimized machine code (Section 3.5), so that the resulting code is dense and can be executed linearly without cache misses (Section 4.2).

All of these innovations result in a system that handles additions, removals and updates at a cost that is proportional to the size of the changes. While we eliminate execution overhead by utilizing adaptive compilation, the runtime performance gain comes at the cost of additional startup time. In Section 4.3 we provide benchmarks for highly dynamic contexts, which demonstrate that our system is significantly faster than existing systems in common cases and performs reasonably well in the worst case.

2 An Incremental Rendering API

The main idea of our extension for rendering systems is the notion of tracking all incremental and structural changes to the objects that need to be rendered. Since existing graphics APIs have no concept of these changes, we need to introduce a new level of abstraction on top of these APIs.

A number of rendering systems (e.g. Inventor [Strauss 1999], OpenSceneGraph [Burns and Osfield 2004], OpenSG [Reiners 2010]) use scene graphs as a useful higher-level abstraction for the rendering intent of the programmer. Naive scene graph rendering simply traverses the scene graph in depth first manner while directly issuing graphics commands to the graphics hardware. Since this simple approach yields unsatisfactory performance, various optimization strategies for scene graphs have been introduced, such as restructuring or compressing the scene graph (e.g. Inventor [Strauss 1999]), or the creation of render caches [Wörister et al. 2013].

We do not mandate a specific high level representation (such as scene graphs), but introduce a new simple representation that is just at the right level of abstraction to be able to programmatically encode incremental changes to the rendering intent (see Figure 1). If scene graphs are used as a high level abstraction, the computation of the dependency graph introduced by Wörister et al. can be used to map the changes from the scene graph to our simpler level of abstraction. Note however, that our representation can deal with structural changes (i.e. additions and removals of content to be rendered), that cannot be handled by Wörister et al.[2013].

The scope of this paper is to optimize interaction with the graphics hardware where we assume high-level semantics already being mapped to a set of drawable objects which we will refer to as *render objects*. Each *render object* is associated with the transitive closure of all its properties required for issuing the appropriate draw call. In the following we shall refer to those properties as *arguments* of the render object.

In order to handle value changes, i.e. changes of arguments of render objects, each parameter needs to be stored in a reference cell, an identifiable cell containing exactly one value at a time. By providing an explicit method for setting a new value, all necessary modifications to internal data structures can be triggered by value changes. Such changes include for example changes of transformations triggered by moving the viewpoint or changes of vertex coordinates triggered by edits, but may also include changes of the visibility state of a render object for temporary culling: this can be implemented by making the execution of the actual draw call dependent on a changeable boolean value.

For structural changes we provide functions for submitting additions and removals of scene geometries. To this end we offer the abstraction of the *render task* which serves as a handle to a set of render objects. Typically a render task is maintained for each render window. Adding and removing render objects to a render task makes it possible to dynamically change the rendered content, and running the render task executes the graphics API calls representing the current set of render objects.

Based on these assumptions we define an abstract incremental rendering API in the following way:

```
class Ref<T> { setValue(T newValue); } // reference cell

class RenderObject {
  Ref<DrawMode> DrawMode;
  Ref<IBuffer> IndexArray;
  Ref<Dictionary<Slot,IBuffer>> VaryingAttributes;
  Ref<Dictionary<Slot,ConstantBuffer>> UniformBuffers;
  /* ... */
}
```

```
interface ITask { void Run(); }

class RenderTask : ITask {
  void Add(RenderObject ro);
  void Remove(RenderObject ro);
}

interface IIncrementalRenderAPI {
  RenderTask NewRenderTask();
  ITask NewClearTask(Ref<Color> col, Ref<float> depth);
  /* ... */
}
```

Note that the API defined so far operates on sets of render objects instead of lists, i.e. it cannot express user-defined draw order. In Section 3.4 we extend our system to also support specified draw orders as required for rendering in the presence of transparency.

3 Adaptive Optimization

It has been observed that reordering drawing code in order to minimize state changes in the graphics API is the most important optimization for achieving high rendering performance (e.g. . Wörister et al. [2013]). In the following section we show how a state sorted representation can be maintained while allowing for incremental and structural changes.

3.1 Overview

Our approach maintains an optimized program of executable machine code *for each render task* at any point in time. Based on the assumption of temporal coherence, i.e. that the scene does not change significantly between successive frames, all optimizations are performed incrementally in running time proportional to the size of the change ($O(\Delta)$). An overview of our approach is shown in Figure 1.

We allow for incremental changes that are triggered by value changes to reference cells, as well as structural changes that add or remove addtional geometry. Startup is performed by adding all initial scene objects as render objects to an intitally empty render task.

On the level of abstract instructions, we perform *state sorting* (see Section 3.2) and *redundant call elimination*. Both optimizations are based on the notion of referential equality of reference cells, i.e. the actual value of a reference cell may change over time, but the identity of the cell (its reference) does not. Therefore we can efficiently identify successive state setters using the same reference cell and optimize to a single one, i.e. the second setter is redundant (see Figure 2).

However, the overhead of interpreting abstract instructions for thousands of objects each frame over and over again is considerable. Therefore, instead of using an interpreter to map abstract instructions to concrete graphics commands [Wörister et al. 2013], we completely eliminate interpretation overhead by further compiling abstract instructions down to executable native machine code representing concrete driver level calls (see Figure 1). Crucially, this compilation is again incremental and triggered by changes in reference cells.

3.2 Generating Efficient Abstract Machine Code

For the moment we focus on unordered sets of render objects. In Section 3.4 we will extend our system to support user-specified order (e.g. back-to-front or front-to-back).

The unordered set of render objects shall be sorted in such a way that the overall number of state transitions becomes minimal. The

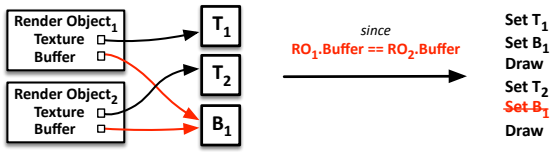


Figure 2: *Under shallow reference equality the second* `Set` *command with the reference to the same buffer is redundant.*

optimal solution for this problem can be formalized as the *Open Traveling Salesman Problem*, whereby each render object corresponds to a node in a graph and the cost of switching state corresponds to the edge labels. Since rendering state consists of multiple sub-states, where each state is associated with a specific transition cost (e.g. setting framebuffers is considered to be more expensive than binding buffers), total cost can be computed using Manhattan distance. Since the cost of switching a particular state can only be estimated and varies between GPU architectures and driver implementations, state sorting does not need to be precise. Thus, solving the *Open Traveling Salesman Problem* which is known to be *NP-complete* is not necessary, especially when considering that additions and removals of render objects should be possible which in general would require a global re-optimization.

In order to support fast updates with low constant overhead, we decided to use a *trie* data structure (see Figure 3) supporting the following operations:

1. Reasonably fast additions and removals of render objects, while maintaining approximate order.
2. $O(n)$ execution, i.e. traversing the data structure shall not introduce additional cost.

Each level of the trie uses a specific rendering state as its key, ordered by the cost of switching the respective state. Currently we use shaders, textures and buffers as keys since they seem to have the highest impact on performance.

At some point, any further refinement of state grouping does not yield additional performance improvements (see Section 4) so instead of storing single render objects in leaf nodes we create buckets containing a set of unordered render objects. The depth d of the trie is the dimension of the state vector suspect to state grouping. Thus, the cost of adding or removing a render object is equivalent to searching the respective bucket in a trie of depth d. Furthermore, when using hash sets for modeling buckets this can be done in constant time (1).

In order to support efficient execution (2), we maintain a linked list of buckets induced by the trie (see blue nodes in Figure 3).

Instead of repeatedly checking redundant calls each frame, we separate optimization and execution by introducing an abstract description of graphics commands similar to Wörister et al. [2013]. In contrast to their approach of optimizing a predefined unit of compilation called render cache, we perform redundancy removal on the fly while generating the abstract machine code. Since each render object is associated with its full state vector, which is guaranteed not to change due to referential equality, we can generate state transition code for each render object given some predecessor render object. The ability to optimize code on the fly is crucial and allows for incremental changes and optimizations without global optimization.

In order to generate abstract machine code for a given render object and its associated state vector we use a function which performs the state transition given the state vector of the previous render object

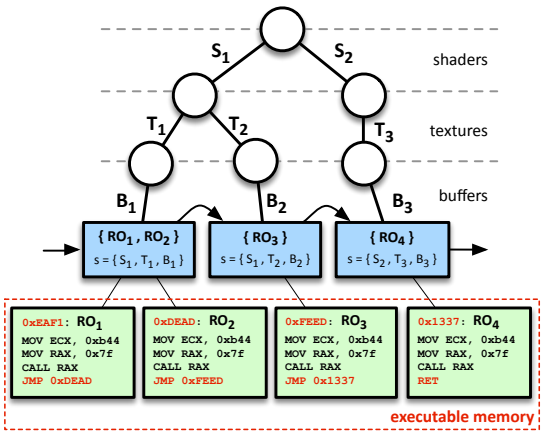


Figure 3: *Bucket trie for state vectors* $(shader, texture, buffer)$. *Leaf nodes (blue) contain a bucket with incomparable render states. Furthermore, each leaf node is associated with* code fragments *for each render object (green). These code fragments directly contain executable machine code which is linked using* `JMP` *instructions.*

depending on the insertion point. In order to efficiently add and remove code for render objects we use a doubly linked list which holds code for specific render objects. Additionally, we maintain an index of contained render objects, which maps render objects to their associated linked list node to allow for additions and removals at arbitrary positions in constant time (see Figure 4).

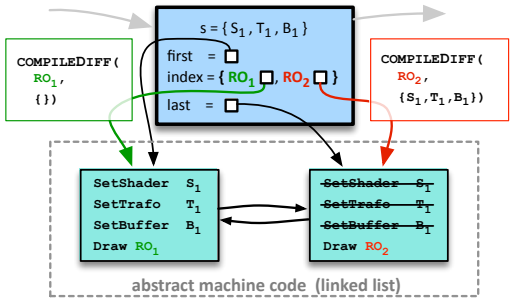


Figure 4: *Render object bucket (leaf in the bucket trie) with per-bucket constant state* (S_1, T_1, B_1). *For each render object we maintain a piece of abstract machine code in a doubly linked list. Code for a specific render object depends on its predecessors output state and can be compiled using* COMPILEDIFF *(see Algorithm 1).*

Furthermore, in order to respond to value changes of render state components adaptively, we register callbacks for each changeable parameter which directly patch the abstract machine code. Each bucket can now be rendered by sequentially executing the optimized abstract machine code while the complete scene can be rendered by traversing the linked list of buckets.

3.3 Incremental Compilation of Abstract Machine Code to Native Machine Code

Although separation of optimization and execution is crucial for performance, interpreting abstract graphics instructions induces significant constant overhead. In early prototypes, we used a hand-tuned interpreter implemented in the C programming language as well as various virtual machine implementation techniques (see Section 4).

Algorithm 1 Emits abstract machine code for render object r starting in render state s.

```
1:  procedure COMPILEDIFF(r, s)
2:      if texture(r) ≠ texture(s) then          ▷ textures differ
3:          ▷ emit to code buffer and register for changes.
4:          EMIT(SETTEXTURE,texture(r))
5:      end if
6:      if indices(r) ≠ indices(s) then          ▷ index buffers differ
7:          EMIT(SETINDEXBUFFER,indices(r))
8:      end if
9:      ...
10:     ▷ emit draw instruction depending on the draw mode
11:     EMIT(DRAW(drawMode(r)))
12: end procedure
```

Regardless of the technique, each instruction still needs to be decoded and dispatched each frame. Therefore, following the recurring theme of this paper to apply optimization to a structure once, in order to avoid repeated overhead, we introduce adaptive compilation of graphics commands:

- Allocate a block of executable memory.
- Traverse the list of render objects once. For each object generate a *code fragment* of machine code calling the appropriate API functions, and put this code fragment into the block of executable memory.

In order to support dynamic changes, we also keep the original trie and its render buckets. In the case of adding a render object, we find its bucket, allocate memory for the corresponding code fragment, and use the memory manager which is filled with the executable code generated by COMPILEDIFF with respect to the the previous render object of this bucket. Similarly to a linked list implementation we need to patch two jump instructions causing the execution to continue in our new code fragment and jumping back when its execution is finished. Removals can be handled by simply skipping the code fragment in question by patching the jump of the predecessor and deleting the corresponding memory region using the memory manager.

Note that these changes to the code fragment list cause the jump instructions to be non-local in general. Therefore the code may become increasingly inefficient when applying further changes (see Figure 7).

For in-place updates, the callback mechanism introduced earlier carries over to native code generation. As a result, value changes of render object arguments directly patch the associated bit code without additional overhead (see Figure 5). As an example consider a 3D modeling application. Whenever the user changes the blend mode for an object, the system automatically responds to this change and specifically generates code for setting up blending accordingly.

3.4 Introducing Drawing Order

In order to render transparent objects using current hardware, it is necessary to sort render objects back to front, and perform the associated render commands in this order. Since this order can change with every frame, our assumption of temporal coherence breaks, and compilation and redundancy removal optimizations on this set of render commands is a wasted effort.

For this reason we resort to a simpler scheme for render objects which require some rendering order:

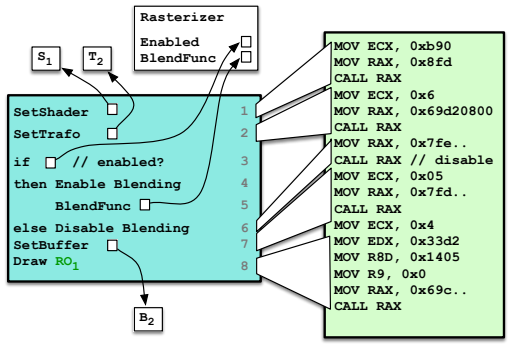


Figure 5: *Abstract machine code (left), assembly code for the draw commands (right). Each virtual machine instruction registers callbacks for all changeable parameters. If the parameter changes, the system automatically patches affected binary code. This applies for structural changes as well. In this example, the machine code disables blending. If blending is finally enabled, additional code for specifying the blend function needs to be patched in.*

- disable redundancy removal for these render objects: render states must be set, since the exact render state is unknown due to the changing order
- annotate these render objects with a key
- sort the corresponding code fragments as necessary

Based on this simple scheme, we can extend the incremental rendering API to handle rendering order. We allow adding render objects with a key, that are subject to changing rendering order, and we provide a function for sorting all render objects with a key, by suppling a callback function that needs to specify the rendering order for each key value:

```
class OrderedRenderTask : ITask {
  void Add(Key key, RenderObject ro);
  void Remove(Key key, RenderObject ro);
  void SortObjects(Func<Key, Order> orderFunction);
}
```

3.5 Defragmentation

Insertions and deletions of code fragments, as well as changes in the code size of code fragments lead to an increasing number of non-local jumps in the sequence of code fragments in executable memory. This negatively impacts cache performance [McFarling 1989], especially for programs with static control flow. In order to improve runtime performance in presence of structural changes, we need to perform defragmentation of the instruction stream. Note that, in contrast to conventional approaches which store optimization structures as data, we do not benefit from compacting garbage collection, as employed by virtual machines [Blackburn et al. 2004].

Therefore we use a similar approach to concurrent garbage collectors to compact code fragments. In order to avoid blocking the execution during defragmentation we introduce a procedure based on microsteps. Execution must be synchronized with those microsteps but not with the entire defragmentation. Thus defragmentation and execution can be interleaved performing only a subset of all defragmentation steps after each frame.

Similarly to the algorithm proposed by Dijkstra et al. [1978], which uses memory fences in order to synchronize evacuation and execution, our algorithm performs synchronization on a per block level where each block is handled by a so called microstep.

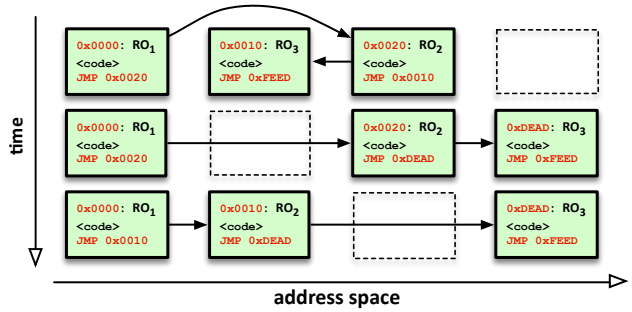


Figure 6: *A defragmentation microstep evacuates code fragment RO_3 in order to have enough space to move the code fragment RO_2 (which follows code fragment RO_1 in execution order) into the correct place to avoid the non-local jump from RO_1 to RO_2.*

A microstep is divided into two phases: (see Figure 6)

evacuate Moves all code fragments consuming the space adjacent to the current one to a new memory location (provided by the memory manager) in such a way that the desired code fragment (the next one in execution order) can be moved there.

compact Actually moves the next code fragment to the free space obtained by the evacuation step and ensures that remaining space is freed using the memory manager (if such a space exists).

Note that this is a very simple example for illustrative purposes. In general the code fragment sizes vary and it may be necessary to `evacuate` more than one code fragment in order to free enough space for the subsequent code fragment. For the evacuation step it is also necessary to allocate new space for those evacuated code fragments.
Note that performing one of these steps may even increase the overall fragmentation but after repeating this step for all existing code fragments the final output will only contain local jumps.

4 Evaluation

4.1 Implementation

Our implementation is written mostly in *F#* and *C#* and runs in *.NET4.5* and *Mono 3.12*. The graphics API we use is *OpenGL 4*. Note that for application-level code we use high-level programming concepts. Performance critical parts however operate on raw memory. Meta data for code fragments and render tasks is stored in managed data structures, while code fragments are placed in executable memory. In our implementation we use a custom assembler (*AMD64*) which generates machine code and associates meta data with memory offsets required to perform in-place updates of machine code.

4.2 Different Virtual Machine (VM) Implementation Techniques

There are several different ways of implementing a virtual machine for our abstract machine code, which span the gamut from perfect execution with poor update performance to perfect update performance with poor execution. In the following we compare the performance of our approach to a series of commonly used techniques. These techniques are:

Direct Threaded Code This implementation serves as a baseline

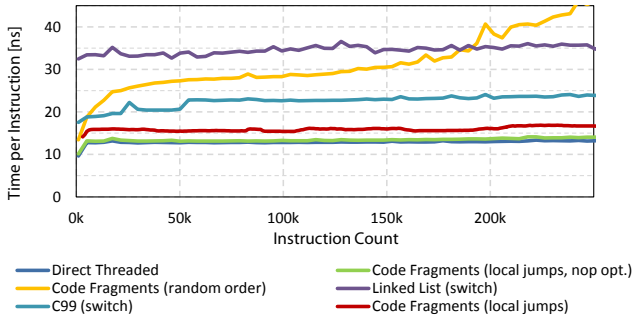


Figure 7: *Comparison of different VM implementations with varying size of programs. Even with increasing program size (instruction bound), all implementations roughly maintain constant execution time per instruction. However, a naive implementation of our code fragment technique becomes significantly slower with large programs when ordered randomly. This is mainly due to cache misses during execution (see defragmentation). In case of non-random programs, i.e.* Code Fragments (local jumps, nop optimization)*, our implementation is as fast as the compiled version using direct threaded code.*

and consists of an unrolled instruction stream of all calls [Bell 1973]. Therefore this approach can be seen as an optimal runtime implementation with very poor update performance.

LinkedList (switch) This represents the most straight-forward way to implement an interpreter for our abstract machine code using a C# switch-statement.

Virtual Methods Since switch based interpreters are usually compiled to jump tables, virtual methods yield similar results to *LinkedList(switch)* and is therefore omitted in the graph.

C99 (switch) Since the managed interpreter spends most of its execution time transitioning from managed to unmanaged code, a simple C99 implementation for the interpreter gives a significant performance gain. This is caused by the fact that there is only a single managed/unmanaged transition when running the interpreter.

Code Fragments (local jumps) Uses our fragments as described in Section 3.3 with all jump distances 0.

Code Fragments (random order) Same as before, but with randomly chosen code fragment execution order. Therefore the jump distances are relatively large (especially for large programs). This motivates defragmentation as described in Section 3.5.

Code Fragments (local jumps, nop optimization) Similar to *Code Fragments (local jumps)*, but all distance 0 jumps are replaced with `NOP` instructions. This is motivated by the observation that "useless" jumps cause measurable overhead.

In our test setup (see Figure 7) we use 25 different functions (native) taking 1 to 5 integer arguments. These functions do not execute any code and return immediately since we want to measure the overhead introduced by the VM. This setup simulates the driver functions used by our real backend implementation and shows that our implementation achieves near minimal execution overhead.

In a second test, we also measured the cost of random insertions into an instruction stream of varying size organized as code fragments while disregarding any associated cost of resource uploading. The result of a nearly constant insertion time of about 6 microseconds for a single insertion, allowing for a theoretical maximum of more than 150,000 insertions per second, poses no practical limitation in real-world scenarios.

4.3 Rendering Performance

Having shown that our system achieves near minimal execution overhead for submitting commands to the graphics driver, we now need to assess the possible speed up when geometry is actually rendered. This possible speed up is clearly dependent on the structure of the scene. There are two extreme cases for this structure:

- Scenes with a high degree of geometric repetition can be rendered using hardware instancing, and the number of actual draw calls is very small. In this case our system will not achieve a lot of speed up, since there is not a lot to optimize.
- Scenes with very little or no geometric repetition require that graphics states need to be set for each object, so again the potential speed up by optimising the submission code is small.

Most realistic scenes will fall in between these two extremes of the spectrum, i.e. there will be some geometric repetition but not for all objects and not with a large number of instances. For our tests we therefore used a static scene of varying number of distinct geometries each replicated twice on average. Each geometry is assigned a texture out of 128 distinct textures randomly. By using generated geometry we ensure that polygon count (and GPU load) remains roughly constant for each test case. Note that, such scenes with huge geometry counts, but different properties are a necessity in applications such as CAD or modelling software where each object needs to be editable individually.

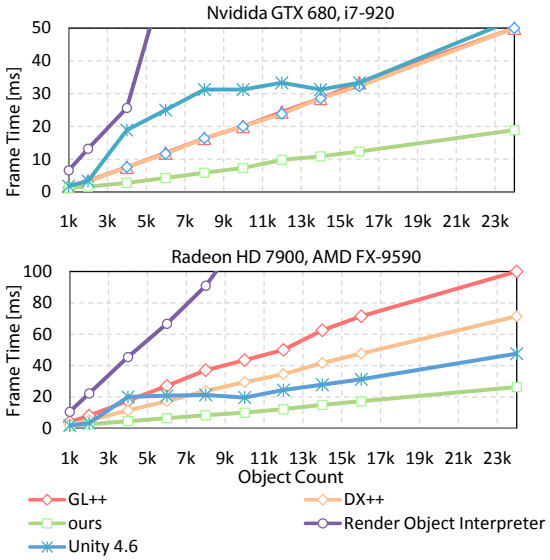


Figure 8: *Scenes of varying object count with roughly constant GPU load. Our compiled approach performs best in all cases and outperforms our Render object interpreter as well as C++ OpenGL/Direct3D11 render object interpreters and Unity 4.6. Note that, due to driver overheads, all configurations are CPU bound.*

We evaluated our approach by comparing it to the following four implementations:

Render object interpreter An implementation in .NET, which uses the same data structures as our compiled variant (proposed method). Essentially this implementation reflects a

runtime variant of COMPILEDIFF which issues commands directly instead of emitting instructions.

GL++ A `std::vector` stores the set of render objects, where each render object contains all prepared GL resources for the object. Each render object contains an additional uniform buffer for storing the per-object model matrix. Note that this implementation issues *exactly* the same OpenGL commands as *our approach* except for state sorting.

DX++ Just like *GL++*, a `std::vector` is used to maintain the list of render objects. Each object contains a separate constant buffer which stores the model matrix.

Unity 4.6 A straight-forward rendering of the same test scenes using the Unity 4.6 rendering engine.

Note that our reference implementations perform runtime redundancy removal, i.e. each graphics command is checked to be effective by comparing the active state for each state attribute (e.g. textures).

We performed out tests under Windows 8 using two different system configurations:

- Nvidida GeForce GTX 680 with 2048MB graphics memory and an Intel i7 920 @ 2.67Ghz and 12GB RAM
- AMD Radeon HD 7900 Series, 3072 MB graphics memory and an AMD FX Eight-Core Processor, 4.7GHz, 16GB RAM

The results are shown in Figure 8. Obviously interpretation of the render objects in the managed .NET implementation (*Render object interpreter*) incurs a significant overhead due to the transition from managed to native environment for each draw call. Comparing to the GL++ implementation we see that our system achieves a significant speed up of about factor 2.5 on the Nvidia system and about factor 4 on the AMD Radeon system. The speed-up when compared to DX++ on the AMD Radeon is lower, indicating that the DirectX driver may be able to perform additional optimizations that are not available via the OpenGL API, still our system outperforms the DX++ implementation on both systems. Notably our system also outperforms or at least matches the performance of Unity 4.6 (which uses DirectX) for all object counts.

It can be seen that our system can achieve significant speed ups in the static case even though it allows dynamic changes (additions and removals of render objects) which cannot be easily handled in other comparable systems (e.g. Wörister et al. [2013]).

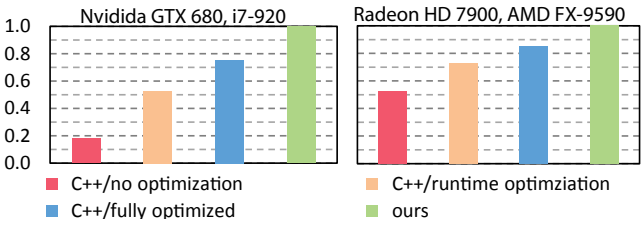


Figure 9: *A comparison of execution performance in frames per second, normalized to 1.0 for our implementation. We show this relative performance, since it was nearly the same across multiple test instances with varying render loads.*

In order to break down the performance improvement of our system, we implemented various variations of our backend:

C++/no optimization A straight-forward implementation using a simple C++ switch statement executing all instructions for each fragment without any pre-filtering. Code is stored in a linked list of fragments where each fragment uses a `std::vector` of instructions. Each instruction is represented by a `struct` containing an `opcode` and appropriate arguments. Note that these fragments are necessary to provide the needed flexibility for our top-level API.

C++/runtime optimization The same implementation as *C++/no optimization*, while additionally maintaining OpenGl state throughout its execution and filtering out redundant calls by checking them against that state.

C++/fully optimized This is the equivalent to our approach but without the JIT compilation, i.e. a tight C++ loop for interpreting state sorted, pre-filtered graphics instructions.

ours Our full implementation with native JIT optimization.

In order to eliminate bias towards a specific implementation we use exactly the same application setup and the same sorting mechanism (i.e. a trie to support fast updates) for all implementations. In fact all approaches except for **C++/no optimization** issue exactly the same OpenGL commands. A relative comparison is shown in Figure 9. The values for the relative benefit were measured using the same synthetic tests as in Figure 8. Since the deviation was very small across multiple instance sizes we just show an average over all different render loads. The results show, that even compared to a fully optimized C++ implementation our approach achieves significant speed up. This validates our approach of compiling the abstract render code to machine code for submitting rendering commands.

By exploiting temporal coherence our system uses the same optimizations for the dynamic case as in the static case. However, due to the necessity of updating all the different representations (see Figure 1), the speed of dynamic changes (additions/removals) necessarily has an upper limit which we need to find out.

To evaluate this, we measured the time per change (addition or removal) for modifying a rendering workload of various size (see Figure 10), in both the compiled *code fragments* implementation and the interpreted *linked list* implementation. Although the cost of changes in the compiled version is significantly higher than in the interpreted version, nevertheless we can perform about 1400 changes per second in the compiled version, which corresponds to adding about 23 render objects per frame at 60 frames per second.

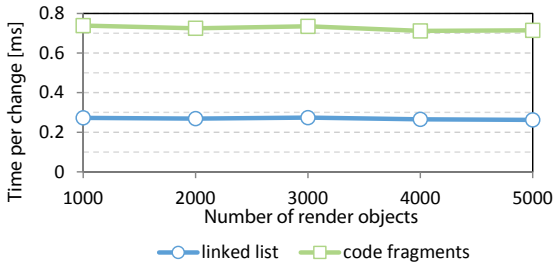
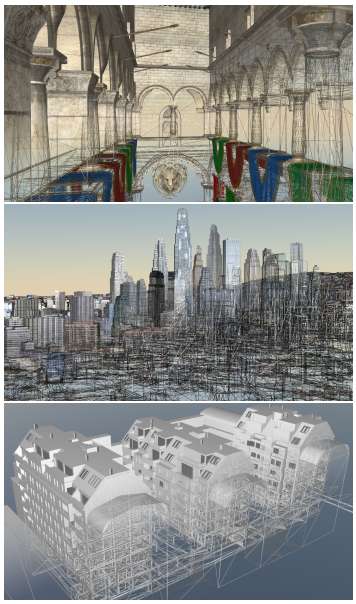


Figure 10: *Structural updates (additions and removals) performed on a scene of varying size (number of render objects). Both implementations provide constant running time per change. In this test we perform batch changes of size 40 which consist of 20 additions as well as 20 removals. In total we can perform about 3500 objects additions/removals per second in the linked list implementation and approximately 1400 in the code fragments implementation. At 60 fps that corresponds to about 23 modifications per frame which seems reasonable for editors as well as games.*

We think that the increased modification cost is well worth the demonstrated significant rendering performance improvement and

is still low enough for typical real world applications such as 3D editors or games.

Note, that for incremental changes of values, the update performance is different, depending on the graphics API we use for submitting the value to the hardware: if the value change is represented in an explicit state object, it can be performed by just changing the value, and uploading the changed value to the hardware. In this case the speed is only limited by the bandwidth to the graphics hardware and our new system has the same performance for changes as the one by Wörister et al. In the other case, when the value change is not represented in an explicit state object, and the value change requires a modification of the associated code fragment, the size of the code fragment can possibly change and the resulting worst case performance is the same as shown for structural changes in Figure 10. Given our premise that there is only little change between successive frames, this represents the maximum possible performance given the limititations of existing graphics APIs.



Sponza24: Crytek's Atrium Sponza scene with 392 objects and 20 textures, replicated 24 times without instancing summing up to 3,452,208 triangles.

HugeCity: A generated city with 6,580 objects and 598 textures and a total of 17,219,220 triangles.

Architecture: A big architectural scene consisting of 7,022 objects summing up to 1,501,606 triangles.

Figure 11: *Scenes used for performance tests.*

4.4 View Frustum Culling

Although we achieve an acceptable performance for structural changes, adding or removing about 1400 objects per second is not fast enough for view frustum culling. As already mentioned in section 2, temporary culling operations can be implemented with value changes. This technique avoids the use of structural updates for temporary culling and thus reserves their use for real changes in the scene such as editing operations in CAD software (i.e. creating or deleting entities) or the creation of characters or objects in games. This separation reflects the necessity to distinguish between high frequency changes that leave their resources allocated, and low frequency changes with proper handling of deallocation. By using this improved technique the dynamic performance of our system is clearly sufficient.

4.5 Real world

As we have shown in previous sections our system efficiently issues commands to the underlying graphics hardware — in fact we practically eliminate all overheads in the graphics backend. However, our performance improvements depend on what is actually rendered. Of course, in scenes with high GPU load, i.e. when the actual rendering is the bottleneck, our system does not provide additional performance. Still, low overhead has other benefits. Otherwise wasted CPU cycles can be used for other computations or just idle waiting for the graphics system, possibly with lower energy consumption. As already indicated in the artificial test scene, our system is not only dependent on scene complexity, but also on the underlying graphics driver system.

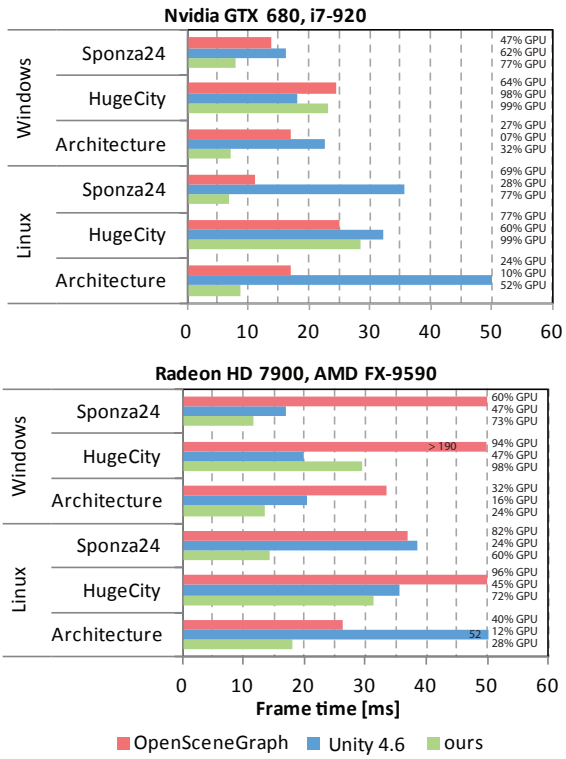


Figure 12: *Our system outperforms Unity 4.6 and OpenSceneGraph 3.0.1 in most cases (see Figure 11 for screen shots of the used scenes). On Windows with the HugeCity scene , we observe slightly inferior performance compared to Unity (which is running DX11). Note that this difference does not appear in our Linux setup (identical hardware), in which both systems use OpenGL. In a significant portion of the tests we successfully made rendering GPU bound (with the exception of the architectural scene) whereas especially Unity suffers from being CPU bound due to overheads when submitting draw commands.*

In order to find out how our system performed on real world rendering loads we compared its performance to Unity 4.6 and OpenSceneGraph 3.0.1 using the three scenes shown in Figure 11. We tested these scenes on both Windows 8 and Linux in the form of Ubuntu 14.04 LTS and with the same system configurations as in our artifical rendering test. The results of this comparison are shown in Figure 12. Our implementation outperforms both systems in nearly all cases, and also increases GPU utilization to the point that rendering is almost entirely GPU bound for a large part of the tested cases.

This result is supported by comparing our compiled *code fragments* implementation with our interpreted *linked list* implementation (see Figure 13): the compiled submission of graphics commands again significantly improves GPU utilization.

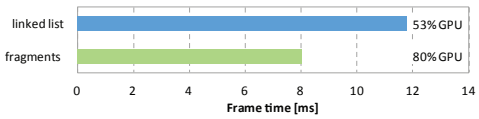


Figure 13: *In this benchmark we use Sponza24 to compare the linked list and code fragments implementation. This shows that our compiled rendering code significantly improves GPU utilization.*

Please note that although we tried the best to make the tests fair, there is still some bias towards specific hardware and driver combinations, simply due to the fact that there often is no single implementation which is equally performant on all platforms. Also note, that on Windows Unity uses DirectX while both other systems use OpenGL.

5 Related Work

There is a trend towards more stateless graphics APIs in order to reduce implicit state. *DirectX 10/11* introduced separate state objects for semantically related subsets of the global state (e.g. RasterizerState, InputLayout). These state objects can be shared between related draw calls, and for dynamic scenes they can be modified without changing the actual graphics commands. Similarly, *OpenGL 3.0* [Segal and Akeley 2008] introduced *Vertex Array Objects* (VAO), which capture all relevant state needed to supply vertex data. Apple's *Metal* [Apple Inc. 2014] provides direct access to graphics state and exposes *command buffers* similar to *OpenCL*'s command queues [Khronos and Munshi 2009], or the Khronos Group's proposed *Vulkan* API [Khronos 2015].

Although stateless APIs provide a mechanism for expressing value changes in render states and draw call parameters, there is no notion of structural changes such as addition or removal of objects to be rendered. Although stateless graphics APIs (e.g. direct state access in OpenGL [Segal and Akeley 2014]), remove the need of setting up appropriate rendering state sequentially, draw commands need to be issued to appropriate command queues. Note that display lists support this kind of batch submission and can be reused in successive frames. However, display lists are static and cannot be changed structurally after creation. Most recently, Nvidia's command list extension [NVidia 2015] introduced mechanisms for compilation of graphics commands and updating graphics states. Although internal implementation details are not publicly available we believe our approaches and findings could benefit from each other.

Today's graphics APIs allow drivers to perform a rich set of low-level optimizations. For example, redundant draw calls can be filtered out early. Unfortunately, due to lack of high-level information, low-level graphics APIs simply cannot perform all possible optimizations. As an example consider the Z-cull optimization in the presence of complex shaders modifying Z-values dynamically. Mantler and Hadwiger [2007] propose to extend graphics drivers to support programmable z bias in order to safe early z optimization, even in the presence of depth modification in pixel shaders. Another example of missing information at driver level is *Level of Detail* (LoD). Cohen et al. [2003] propose to solve LoD at the driver level. The same applies to geometric mesh optimizations such as generation of triangle strips or other complex optimizations aiming for vertex locality and reduced overdraw on a per mesh granularity [Nehab et al. 2006].

Thus, for maximum performance, graphics programmers need to manually tune graphics commands for specific scenes or implement optimizations at application or graphics engine level. Graphics engine optimizations go back to early work in the field of scene graphs. Strauss [1999] optimizes scene graphs by applying persistent transformations to the input scene graph (e.g. pulling up costly changes). Other optimizations based on **preprocessing** include automatic pre-transformation of geometry, packing of geometry or textures or merging of slightly different materials. Notably, those optimizations can be found in graphics programming and scene graph toolkits such as *OpenSceneGraph* [Burns and Osfield 2004], *OpenSG* [Reiners 2010], or *IRIS Performer* [Rohlf and Helman 1994]. However, the key problem with preprocessing is the loss of dynamism. For example, packed or pre-transformed geometries cannot be changed individually or at least need to be de-optimized prior to modification.

Another optimization introduced in the context of scene graphs is **caching**. In their scene graph system Durbin et al. [1995] associate each scene graph node with a cache containing all render commands which are required for rendering the sub-graph contained by the node. The advantage of generating such caches is twofold: Firstly, the rendering system executes the prepared cache instead of costly traversing the sub-graph. Secondly, the cache can be optimized in various ways. Optimally, the cache contains a *streamlined* array of graphics commands, which additionally can be optimized for faster execution, e.g. by removing redundant graphics commands. Optimizing instruction streams for fast execution is by no means limited to computer graphics. Hirzel et al. [2014] provide a comprehensive overview on stream optimizations in other fields of computer science.

One key problem with rendering caches is consistency with the represented scene. Changes in the original scene graph need to be mapped to changes in the optimized structure efficiently. Based on recent work on incremental evaluation in the computer language community Wörister et al. [2013] introduce rendering caches which can be updated incrementally. They use a dependency graph in order to keep the rendering cache consistent with the original input scene data structure.

They show that render caches serve as a solid basis for various optimizations and identified *Redundancy removal* and *State sorting* to be the most effective in spite of the fact that these optimizations are also usually performed on the fly in the driver [Frascati and Seetharamaiah 2014]. Unfortunately these optimizations—when performed on rendering caches—have runtime complexity $O(n)$, where n is proportional to the number of objects to be rendered. And even more detrimental, each addition or removal of a render object requires the render cache to be recomputed, i.e. almost identical optimizations are performed over and over again.

6 Discussion

Our system is based on the same assumptions as the one by Wörister et al. [2013], namely that a lot of recurring optimizations can be avoided by incremental computation. Although they show how to build a dependency graph in order to track changes originating in a scene graph, their implementation cannot handle structural changes of the rendering workload, i.e. they cannot incrementally handle what we call additions and removals of render objects and their system is limited to value changes in their rendering caches.

In contrast we concentrated our effort on the rendering API without dealing with scene graphs. We show how to deal with structural changes and provide the same type of optimizations for dynamically changing scenes that are normally only performed on static parts of the scene. Additionally we completely eliminate overheads by compiling abstract render code to native machine code.

In their optimizing compiler for rendering assets, Lalonde and Schenk take a similar approach [Lalonde and Schenk 2002] as we

do. They use *render methods* which describe shaders and their input in order to compile and run optimized bytecode in an interpreter. Our paper however, focuses on maintaining a fast render program incrementally. Notably, in contrast to our approach, they use an interpreter instead of native machine code. In combination with our defragmentation scheme, native compilation achieves significant speedups, at least on our target platforms.

Currently our incremental rendering API is implemented on top of traditional stateful graphics APIs, and due to the limited availability could not be implemented and tested against modern APIs (e.g. *Vulkan* API [Khronos 2015], *Metal* [Apple Inc. 2014]) which aim to be more stateless. A part of our effort consists of simulating a stateless API, and will not be necessary with these new APIs, but optimization still needs to be performed on the command stream of a stateless API, just that the actual optimization should be performed in the driver not in the rendering engine. Our paper covers a large part of the design space for these optimization tasks, and thus is highly relevant for the implementation of drivers for stateless APIs.

7 Conclusion

In this paper we introduced an incremental rendering VM with a a novel incremental rendering API that we implemented on top of existing graphics APIs. We show that our VM maintains a fully compiled near-optimal representation of the rendering content at all times, leading to significant performance gains in typical rendering scenarios. Additionally we demonstrate, that the associated increased modification cost for incremental changes is low enough for using this new rendering architecture in real-world applications. Our paper shows that it is possible to exploit temporal coherence in the underpinnings of rendering applications by amortising optimization of rendering instructions over multiple frames.

We hope that our work will influence the design of future rendering APIs and drivers in order to further improve rendering performance.

Acknowledgements

The competence center VRVis is funded by BMVIT, BMWFJ, and City of Vienna (ZIT) within the scope of COMET – Competence Centers for Excellent Technologies. The program COMET is managed by FFG. This research was also partly funded by project "Replicate" (FFG FIT-IT, project no. 835948).

References

APPLE INC., 2014. Metal Programming Guide. https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/.

BELL, J. R. 1973. Threaded Code. *Commun. ACM 16*, 6, 370–372.

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004. Myths and Realities: The Performance Impact of Garbage Collection. *SIGMETRICS Perform. Eval. Rev. 32*, 1 (June), 25–36.

BURNS, D., AND OSFIELD, R. 2004. Open Scene Graph A: Introduction, B: Examples and Applications. In *Proc. of the IEEE Virtual Reality 2004*, IEEE Computer Society, Washington, DC, USA, VR '04, 265–.

COHEN, J., LUEBKE, D., DUCA, N., AND SCHUBERT, B. 2003. GLOD: A Driver-level Interface for Geometric Level of Detail. In *ACM SIGGRAPH 2003 Sketches & Applications*, ACM, New York, USA, SIGGRAPH '03, 1–1.

DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. 1978. On-the-fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM 21*, 11 (Nov.), 966–975.

DURBIN, J., GOSSWEILER, R., AND PAUSCH, R. 1995. Amortizing 3D Graphics Optimization Across Multiple Frames. In *Proc. of the 8th Annual ACM Symp. on User Interface and Software Technology*, ACM, New York, USA, UIST '95, 13–19.

FRASCATI, C., AND SEETHARAMAIAH, A., 2014. Reordering of Command Streams for Graphical Processing Units (GPUs), July 3. WO Patent App. PCT/US2013/068,709.

HIRZEL, M., SOULÉ, R., SCHNEIDER, S., GEDIK, B., AND GRIMM, R. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv. 46*, 4 (Mar.), 46:1–46:34.

KHRONOS, AND MUNSHI, A., 2009. The OpenCL Specification Version: 1.0 Doc. Rev.: 48. https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf.

KHRONOS, 2015. Vulkan, The Khronos Group Inc. https://www.khronos.org/vulkan. Accessed: 2015-04-17.

LALONDE, P., AND SCHENK, E. 2002. Shader-driven compilation of rendering assets. In *Proc. of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '02, 713–720.

MANTLER, S., AND HADWIGER, M. 2007. Saving the Z-cull Optimization. In *ACM SIGGRAPH 2007 Posters*, ACM, New York, USA, SIGGRAPH '07.

MCFARLING, S. 1989. Program Optimization for Instruction Caches. *SIGARCH Comput. Archit. News 17*, 2 (Apr.), 183–191.

NEHAB, D., BARCZAK, J., AND SANDER, P. V. 2006. Triangle Order Optimization for Graphics Hardware Computation Culling. In *Proc. of the 2006 Symp. on Interactive 3D Graphics and Games*, ACM, New York, USA, I3D '06, 207–211.

NVIDIA, 2015. NV_command_list. http://developer.download.nvidia.com/opengl/specs/GL_NV_command_list.txt. Accessed: 2015-04-17.

REINERS, D., 2010. OpenSg: Features Performance. accessed online on 2014-01-14. http://www.opensg.org/wiki/FeaturesPerformance.

ROHLF, J., AND HELMAN, J. 1994. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-time 3D Graphics. In *Proc. of the 21st Annual Conf. on Comp. Graph. and Interactive Techniques*, ACM, New York, USA, SIGGRAPH '94, 381–394.

SEGAL, M., AND AKELEY, K., 2008. The OpenGL Graphics System: A Specification (Version 3.0). https://www.opengl.org/registry/doc/glspec30.20080811.pdf.

SEGAL, M., AND AKELEY, K., 2014. The OpenGL Graphics System: A Specification (Version 4.5). https://www.opengl.org/registry/doc/glspec45.core.pdf.

STRAUSS, P., 1999. System and Method for Optimizing a Scene Graph for Optimizing Rendering Performance. Patent. US 5896139 A.

WÖRISTER, M., STEINLECHNER, H., MAIERHOFER, S., AND TOBLER, R. F. 2013. Lazy Incremental Computation for Efficient Scene Graph Rendering. In *Proc. of the 5th High-Performance Graph. Conf.*, ACM, NY, USA, HPG '13, 53–62.