Reducing State Changes with a Pipeline Buffer[*]

Jens Krokowski[†] Harald Räcke[‡] Christian Sohler[†] Matthias Westermann[§]

Abstract

A limiting factor in the performance of a rendering system is the number of state changes, i.e., changes of the attributes material, texture, shader program, etc., in the stream of rendered primitives. We propose to include a small buffer between application and graphics hardware in the rendering system. This pipeline buffer is used to rearrange the incoming sequence of primitives on-line and locally in such a way that the number of state changes is minimized. This method is generic; it can be easily integrated into existing rendering systems.

In our experiments a pipeline buffer reduces the number of state changes by an order of magnitude and achieves almost the same rendering time as an optimal, i.e., presorted, sequence without pipeline buffer. Due to its simple structure and its low memory requirements this method can easily be implemented in software or even hardware.

1 Introduction

In current rendering systems a number of different techniques are integrated to reduce the time to render a 3D scene. View frustum culling, approximation, and occlusion culling are used to reduce the number of primitives that have to be rendered. Another significant factor for the rendering time is the number of state changes performed by the graphics hardware during the rendering process. Such a state change occurs when two subsequently rendered primitives differ in their attribute values, i.e., in their material, texture, or shader program. These state changes slow down the rendering system.

[*]This work is partially supported by the DFG grant ME 872/8-2 and WE 2842/1-1.

[†]Heinz Nixdorf Institute and Institute of Computer Science, University of Paderborn, Germany, Email: {kroko, csohler}@upb.de

[‡]Computer Science Department, Carnegie Mellon University, USA, Email: harry@andrew.cmu.edu

[§]Computer Science Department, Dortmund University, Germany, Email: mw@ls2.cs.uni-dortmund.de

Therefore, in some applications, e.g., in computer games, the primitives are organized in blocks of equal material, texture, and shader program in order to minimize the number of state changes in the graphics hardware. However, frustum and occlusion culling usually require a spatial sorting of the primitives. This stands in partial conflict with the requirement to sort the primitives by attribute value, which means that, in general, this approach leads to suboptimal visibility culling. Further problems occur in interactive applications, where a preprocessing is not possible.

1.1 Contributions

To overcome these problems, we propose a different technique to reduce the number of state changes: A pipeline buffer is included between application and graphics hardware and reduces state changes on-line instead of doing a preprocessing. Therefore, the application can benefit from any (especially spatial) sorting without increasing the running time due to too many state changes.

Our approach is based on a small buffer (storing less than hundred references) and a well chosen selection strategy to rearrange the incoming sequence of primitives on-line in such a way that the number of state changes is minimized.

This method is generic, i.e., it can be used for different types of state changes, and it can be integrated into existing rendering systems. Due to its simple structure and its low memory requirements this method can easily be implemented in software or even hardware.

In our experiments each scene is stored in an octree and we consider the sequence of primitives that results from an in-order traversal of that tree. For such a sequence we typically get that our method

- reduces the number of state changes by an order of magnitude,
- reduces the rendering time by roughly 30%,

- achieves almost the same rendering time as an optimal, i.e., presorted, sequence without pipeline buffer.

Note that these results heavily depend on the type of state changes, i.e., on the cost associated with the state changes.

2 Related Work

Conventional approaches to reducing state changes usually use some kind of preprocessing in order to sort the sequence of primitives with respect to their attribute values. However, these methods cannot be easily combined with culling techniques used in modern rendering systems since these techniques usually require a spatial sorting of the primitives as opposed to a sorting by attribute value. Although this trade-off is known for a long time only very little work is done on paying attention to both factors simultaneously.

The IRIS Performer toolkit [11] resorts the visible geometry by modes at the end of the CULL traversal for each frame and sends geometry and graphics commands to the graphics subsystem in the subsequent DRAW traversal. However, the computation of such a rearrangement is so expensive that this can only amortize if it is used for several frames.

The system of Lalonde and Schenk [8] optimizes the geometric data of video games for the targeted hardware (e.g., PS2, XBox and GameCube) in a preprocessing step using shader programs intensely. In addition, the data is partitioned into smaller packets to fulfill hardware restrictions. These packets are sorted to minimize expensive state changes. This method is applicable for video games with fixed topology of the graphical elements but cannot be used for modeling applications where the geometry is changed dynamically.

Buck et al. [3] provide a state tracking system for the remote rendering system WireGL. It performs lazy state updates to transmit less state data over the network but, in contrast to our approach, it does not alter the sequence of the rendered geometry.

Aila et al. notice also the trade-off between state and spatial sorting. For state sorted applications they propose to add a delay stream in the graphics hardware between the vertex and pixel processing unit [1]. The primitives in the delay stream are used to generate culling information. This way primitives residing in the delay stream may be culled by primitives that were submitted afterwards. The delay stream stores 50K-150K triangles and therfore is several orders of magnitude larger compared to the pipeline buffer.

Spatial sorting applications. Occlusion culling methods compute the visible primitives either during preprocessing and store them in state sorted Potential Visible Sets (PVS) or determine them online. If the scene contains mobile primitives (especially mobile occluders), PVS methods normally fail and, in addition, the memory requirements of PVS methods can turn out to be a problem [5].

On-line methods usually store the scene in a hierarchical data structure, e.g., octree [7], kd-tree [6], or bounding box hierarchy [14], to compute the point-based visibility. During a traversal of such a hierarchical data structure, the visible primitives are classified and only these primitives are sent to the graphics hardware. These approaches may generate a sequence of primitives that contains many state changes. We will show how to improve the rendering time for these scenarios by reducing the number of state changes with a pipeline buffer.

Theoretical analysis. The pipeline buffer scenario is an application of the on-line scheduling problem for sorting buffers [10]. The following is a conversion of this theoretical model to the pipeline buffer application: An input sequence of items which are only characterized by a specific attribute has to be rearranged by a sorting buffer which is a random access buffer with storage capacity for k items. The goal is to minimize the number of state changes, i.e., the number of maximal subsequences of items containing only items with the same attribute value.

In this model on-line strategies are evaluated theoretically in a competitive analysis. In such a worst-case analysis the cost of an on-line strategy, which has no knowledge about the future input sequence, is compared with the cost of an optimal off-line strategy, which knows the whole input sequence in advance. Note that an optimal off-line strategy has also to use the sorting buffer to rearrange the input sequence, i.e., in general an optimal off-line strategy can not just sort the whole input sequence.

It is proven that several standard strategies are unsuitable for this theoretical problem [10], i.e., the competitive ratio of the First-In-First-Out and Least-Recently-Used strategy is $\Omega(\sqrt{k})$ and the competitive ratio of the Most-Common-First strategy is $\Omega(k)$. Further, the Bounded-Waste strategy is presented and it is proven that this strategy achieves a competitive ratio of $O(\log^2 k)$ [10]. Note that the competitive ratio of $O(\log^2 k)$ does not give good performance guarantees for small buffer size k (this is the interesting case in our pipeline buffer scenario) because in this case $\log^2 k$ is quite large in comparison to k and the constant factors in the O-notation have a dominating impact.

3 The Pipeline Buffer

In the following, we call the states that occur during rendering the *attributes* of the geometry, e.g., material, texture, vertex and fragment programs. A *primitive* has the same attribute values all over the described geometry. Hence, primitives can be, e.g., points, triangles, quads or indices to vertex arrays as long as the attribute value managed by our pipeline buffer does not change during the rendering of that primitive.

When a sequence of primitives is rendered, the graphics hardware has to perform certain actions each time the attribute values of two subsequent primitives differ. For example, when two subsequent primitives have different textures or vertex programs the new texture or vertex program has to be loaded before the primitive can be rendered. This process is called a *state change* of the graphics hardware. Moreover, OpenGL performs a validation step to update the internal caches [12].

The rendering time of a sequence of primitives depends significantly on the number and the kind of state changes in this sequence. Changing the color or the blend mode is usually cheap. However, certain state changes, e.g., textures, cause page faults in the internal caches and therefore lower the performance of current graphics hardware. We concentrate on expensive states that are widely used in practice and often changed during the rendering of one frame, whereas the pipeline buffer method does not depend on nor is limited to these states.

The pipeline buffer is easy to integrate in an existing high level rendering system on application level. The only requirement is that each object has a reference to its attribute values. The more general solution we propose is to implement the buffer inside a device driver.

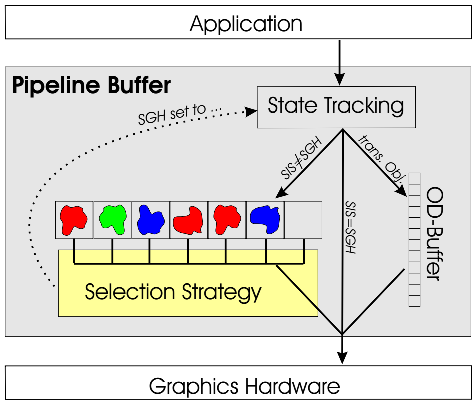


Figure 1: The pipeline buffer as application-independent add-on to the graphics API.

Method. For simplicity, we assume that there is only one attribute shared by all primitives. In Section 3.2 we discuss how to deal with more than one attribute.

The diagram of Figure 1 shows our *pipeline buffer* which consists of a state tracking system, a selection strategy and a buffer with random access that stores up to k primitives whose attribute value differs from the one currently processed by the graphics hardware.

The system catches the state setting commands from the application and records them as the *state of the input stream (SIS)*. If the current *state of the graphics hardware (SGH)* is the same as the SIS we pass all subsequent geometry immediately to the graphics hardware. Otherwise, the geometry is appended at the end of the buffer together with the SIS information. In the case of textures or vertex/fragment programs this is easily possible because a change involves sending a corresponding ID. Material information is stored once in the buffer with a generated ID and all primitives sharing this attribute value refer to this.

In case of a buffer overflow we must evict primitives from the buffer. (A buffer overflow means that k primitives are stored in the buffer.) This forces a state change in the graphics hardware and we have to select the next state. This is done by the *selection strategy*. Several strategies are presented and discussed in Section 4. Finally, all primitives in the buffer that provide the selected state are sent to the graphics hardware.

At the end of the input stream or if the *flush* command is executed the selection strategy determines the next state and all primitives providing it are rendered. This is repeated until the buffer is empty.

3.1 Order-Dependence Buffer

Most graphics APIs require that transparent objects must be blended in back-to-front order. For scenes with transparent objects we propose an additional fixed size FIFO buffer called *order-dependence buffer (OD-buffer)* that stores all transparent primitives to avoid reordering.

If the OD-buffer is full, first the pipeline buffer is flushed and then the OD-buffer is emptied. Finally, when all opaque primitives have been rendered the remaining transparent primitives of the OD-buffer are rendered. This way we achieve that the relative order of transparent objects is not altered and the opaque objects are not delayed relatively to the transparent ones. Order-independent transparency methods like the A-buffer algorithm [4] or the R-buffer [13] present an other possibility to handle this problem.

In-Order Execution. Several graphics APIs, e.g., OpenGL [2] and DirectX [9], specify that primitives mapping to the same pixel must be rendered in the order they were submitted. Reordering the primitives violates this rendering semantic and could cause temporal flickering problems in some special cases like edge high lighting, stencil operations or if identical geometries with different textures are rendered.

Our experience is that for most scenes this affects no or only a small subset of primitives. Additionally, the order dependency can mostly be reduced to a small number of blocks of primitives. Only the order of these blocks is important but reordering the primitives of one block does not affect the final image. For example, for edge high lighting the primitives can be divided in two blocks: Triangles and lines. Therefore, the pipeline buffer can be used to reduce the state changes within one block but must be flushed between the blocks.

In our opinion the in-order dependency cannot be identified fully automated in all cases. Thus, the pipeline buffer can be manuelly flushed or disabled completely by the programmer at the expense of loosing the benefits from the buffer.

3.2 Multiple Attributes

In many applications the primitives have multiple attributes. In this case we can either consider only the most expensive attribute or we can use a chain of pipeline buffers, one for each attribute. In this case we propose to order the attributes increasingly according to their influence on the performance of the rendering system.

For example, let us consider the attributes texture (expensive) and color (less expensive). In this case we first process all primitives in a color buffer and finally in a texture buffer. The reason for this ordering is that a block of primitives created by the first pipeline buffer may be destroyed by a later one.

4 Selection Strategies

In this section, we present the different selection strategies that are implemented and evaluated in our pipeline buffer scenario. In case of a buffer overflow, the selection strategy selects the next attribute value processed by the graphic hardware from the primitives currently stored in the pipeline buffer.

Because our pipeline buffer scenario is related to caching, we consider the standard caching strategies First-In-First-Out (FIFO) and Least-Recently-Used (LRU) in Section 4.1 and 4.2, resp. The Most-Common-First strategy, presented in Section 4.3, is a fairly natural strategy in the pipeline buffer scenario. Finally, we introduce the Round-Robin (RR) strategy in Section 4.4.

4.1 First-In-First-Out (FIFO)

The FIFO strategy assigns time stamps to each attribute value stored in the pipeline buffer. Initially, the time stamps of all attribute values are undefined. When a primitive is stored in the pipeline buffer the FIFO strategy checks whether the attribute value of the primitive has an undefined time stamp. In this case, the time stamp of this attribute value is set to the current time. Otherwise, it remains unchanged. At each buffer overflow the FIFO strategy selects the attribute value with the oldest time stamp and resets its time stamp to undefined.

The FIFO strategy is a very simple selection strategy that does not analyze the stream of primitives. The pipeline buffer acts like a sliding window over the stream of primitives in which primitives with the same attribute value are combined.

4.2 Least-Recently-Used (LRU)

Similar to FIFO, the LRU strategy assigns time stamps to each attribute value stored in the pipeline buffer. Initially, the time stamps of all attribute values are undefined. When a primitive is stored in the pipeline buffer the time stamp of its attribute value is set to the current time. At each buffer overflow the LRU strategy selects the attribute value with the oldest time stamp and resets its time stamp to undefined.

The LRU strategy tries to benefit from the past. However, the LRU strategy and also the FIFO strategy tend to remove primitives too early from the pipeline buffer. Hence, both strategies cannot build large blocks of primitives with the same attribute value, if additional primitives with the same attribute value arrive later in the stream.

4.3 Most-Common-First (MCF)

The MCF strategy is a fairly natural strategy in the pipeline buffer scenario. In case of a buffer overflow, the MCF strategy clears as many locations as possible in the pipeline buffer, i.e., it selects an attribute value that is most common among the primitives currently stored in the pipeline buffer.

The MCF strategy also fails to achieve good performance guarantees since it keeps primitives with a rare attribute value in the pipeline buffer for a too long period of time. This behavior wastes valuable storage capacity that could be used for efficient buffering otherwise.

4.4 Round-Robin (RR)

The RR strategy is a very practical variant of the Bounded-Waste (BW) strategy which is introduced and analysed in a theoretical model [10] (for details see Section 2). Thus, we present first the BW strategy. In the previous sections we saw that a good selection strategy should have the following two properties:

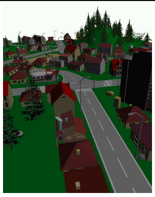
- On the one hand, no primitive should be kept in the pipeline buffer for a too long, possible infinite, period of time.
- On the other hand, there is a benefit from keeping a primitive in the pipeline buffer, if additional primitives with the same attribute value arrive in the near future.

The BW strategy provides a trade-off between the space wasted by primitives with the same attribute value and the chance to benefit from future primitives with the same attribute value. A waste counter is assigned to each attribute value stored in the pipeline buffer. Informally, the waste counter for an attribute value v is a measure for the space that has been wasted by all primitives with attribute value v currently stored in the pipeline buffer. Initially, the waste counters of all attribute values are set to zero. In case of a buffer overflow, the BW strategy increases the waste counter of each attribute value v by the number of primitives with attribute value v currently stored in the pipeline buffer. Then the attribute value with maximal waste counter is selected and this waste counter is reset to zero.

Even if the BW strategy achieves the minimal number of attribute changes in our experimental evaluation, the computational overhead of the BW strategy is relatively large. Hence, we designed more efficient versions of the BW strategy.

A randomized version of the BW strategy is the Random-Choice (RC) strategy. The RC strategy chooses a primitive uniformly at random from all primitives currently stored in the pipeline buffer and selects the attribute value of this primitive. Note that the RC strategy can also be seen as a randomized version of the MCF strategy, i.e., the randomized version of the BW and MCF strategy are identical. However, in the deterministic case, the BW strategy clearly outperforms the MCF strategy. Even if the RC strategy is much simpler than the BW strategy, the generation of random numbers requires a lot of runtime.

Finally, we choose the Round-Robin (RR) strategy which is a very efficient variant of the RC strategy. The RR strategy uses a selection pointer to select an attribute value. Initially, the selection pointer points to the first slot of the pipeline buffer. In case of a buffer overflow, the RR strategy selects the attribute value of the primitive the selection pointer points to. Then the selection pointer is shifted in round robin fashion to the next slot in the pipeline buffer. We believe that the RR strategy still has the same properties on typical input sequences as the BW strategy. In our experimental evaluation, the RR strategy achieves almost the same number of attribute changes as the BW strategy.

Scene	Town	Aphrodite	Bear	Elephant	PowerPlant
					
# Triangles in view frustum	211589	178068	71384	65296	333585
# Textures (mean size)	430 (128^2)*	8 (512^2)	14 (1024^2)	11 (512^2)	17 (mat.)**
# state changes (octree generated)	40341	27395	9215	11171	7529

remaining state changes										
buffer size	10	30	10	30	10	30	10	30	10	30
RR	11744	7473	5753	2908	2025	1138	2593	1475	3331	2047
MCF	17009	12883	6536	3317	2371	1492	2997	1654	3521	2125
LRU	35862	29465	10539	7748	5613	2904	6103	3206	4358	3286
FIFO	37092	34048	12086	10244	4283	4779	6075	4728	4543	3332

remaining state changes in per cent										
RR	29.1%	18.5%	21.0%	10.6%	22.0%	12.3%	23.2%	13.2%	44.2%	27.2%
MCF	42.2%	31.9%	23.9%	12.1%	25.7%	16.2%	26.8%	14.8%	46.8%	28.2%
LRU	88.9%	73.0%	38.5%	28.3%	60.9%	31.5%	54.6%	28.7%	57.9%	43.6%
FIFO	91.9%	84.4%	44.1%	37.4%	46.5%	51.9%	54.4%	42.3%	60.3%	44.3%

Table 1: Statistical overview for different test scenes. State change reductions for buffer size $k = 10, 30$. *Material attributes are replaced by monochromatic textures. **PowerPlant uses only material attributes. *Aphrodite, Bear, Elephant* courtesy of Polygon Technology, Darmstadt (www.polygon-technology.de). *PowerPlant* courtesy of the Walkthrough Group at the University of North Carolina at Chapel Hill (www.cs.unc.edu/˜walk).

5 Implementation and Results

For our experiments we have implemented a prototype rendering system in C++ using OpenGL routines for the rendering. The pipeline buffer is implemented as a linear array and contains references to primitives, e.g., GL_POINTS, GL_TRIANGLES or indices to vertex arrays. All measurements are done using compiled vertex arrays. Primitives are inserted at the smallest free index. Before a primitive is deleted we swap it with the primitive with the largest index. Since our buffer is rather small no other data structures are needed. When a buffer overflow occurs we first find one primitive that is evicted according to our selection strategy. Then we check for every primitive stored in the buffer, whether it has the same attribute value as the evicted one. If it has the same attribute value, we also evict it. Otherwise, it stays in the buffer.

To speed up the comparison of attribute values we assign to each attribute value a unique integer ID. In the case of textures we use the TexID from OpenGL. This way, we can compare two attribute values by a simple integer comparison.

Experimental Setting. Our experiments are made on a Linux based system with a 2GHz Pentium IV processor and an NVIDIA GeForce4 Ti graphics card. Our test scenes and their characteristics are summarized in Table 1. In the Town scene we replaced material attributes by monochromatic textures to generate a test scene with a few hundred different textures. In all our scenes the primitives are given in world coordinates and stored in an octree. To generate the input sequence we traversed this octree by an in-order traversal.

We implemented the four strategies discussed in Section 4 and also the BW and RC strategy that have been discussed in Section 4.4. Our experiments confirmed that BW, RC, and RR are essentially equivalent in their performance with RR having an edge on the rendering time because of its simplicity. Therefore, in our presentation we selected RR as representative strategy. We remark that BW achieves up to 3% higher reduction of state changes.

We made experiments for each combination of strategy, scene, and buffer size. Since there is some redundancy in the results we do not give figures for

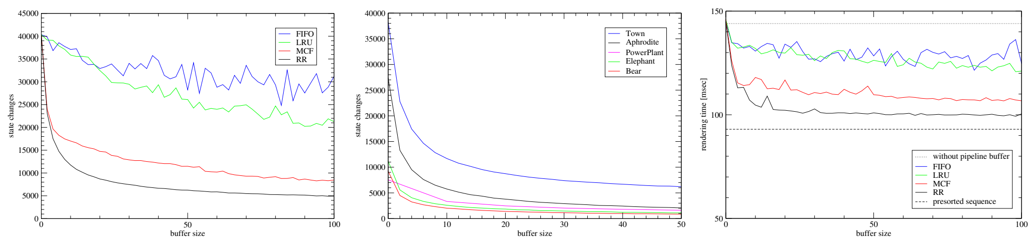


Figure 2: Reduction of state changes: (a) Comparison of the selection strategies (test scene: Town) (b) RR strategy for our test scenes. (c) Rendering time for the Town scene with varying buffer size. The dotted and dashed line represent the rendering time with no and with optimal state change reduction, respectively.

all possible combinations. Instead we try to focus and explain the behavior on selected examples.

In Section 5.1 we compare our selection strategies w.r.t. their reduction of state changes. Then we give more detailed results for RR. Finally, in Section 5.2 we discuss the rendering performance of our strategies, again giving a more detailed discussion for RR.

5.1 State Change Reduction

First we compare our selection strategies w.r.t. their reduction of state changes. We start with the Town scene. In Figure 2(a) we see the number of state changes achieved by the different strategies and for different buffer sizes. The sequence of primitives generated by the in-order octree traversal has 40341 state changes. The highest reduction is achieved by the RR strategy which reduces the number of state changes to 7473 with buffer size 30. The second best strategy is MCF with 12883 changes with buffer size 30 followed by LRU and FIFO. Additionally, the performance of LRU and FIFO strongly depends on the buffer size: A slight variation in the buffer size may increase or decrease the number of state changes significantly. The other strategies are more robust concerning this phenomenon.

For the other scenes the characteristics of the curves are similar to Figure 2(a). However, the relative performance of the strategies depends on the scenes. Therefore, we present the full set of curves only for RR (see Figure 2(b)). For all other strategies we just give results for buffer sizes 10 and 30 for the other test scenes in Table 1.

It turns out that for some scenes the performance of MCF is closer to RR than for the Town scene. This can be explained by the fact that the Town scene contains some textures that rarely occur in the scene. Hence, MCF does not evict primitives with these rarely occurring textures from the buffer for a long time. These primitives block valuable buffer space and this has a negative effect on the performance of MCF on the Town scene.

Even for small buffer sizes a significant reduction is achieved. Our investigation unfolds that most spatial sorting creates long sequences of one attribute value disconnected by single different values, like `AAA B AAAAAA AAAA C AAAAAAAAAAAA B AAAAAAAAA BB AAAAAAAAAA`. Even with a buffer size of $k = 3$ the resulting sequence, e.g., `AAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAA BBBB C AAAAAAA AAA` contains less than half the state changes.

Summarizing we can see that RR clearly outperforms all other strategies w.r.t. to the reduction of state changes. Even though MCF achieves comparable results for some scenes it can be away by more than a factor of 1.5 as the Town scene shows.

5.2 Performance

Since all strategies are fairly simple, it is no surprise that the reduction of state changes results into faster running times. As an example, Figure 2(c) illustrates the rendering time for the four selection strategies on the Town scene. We also give the rendering time without pipeline buffer on the input sequence (generated by octree traversal) and on a presorted sequence, which is the optimum performance we can achieve. It can be seen that RR achieves the best performance among our strategies improving between 28.8% and 35% on the rendering time without sorting buffer for textured scenes (see Table 2). We also observe that the performance of RR is very close to the performance of a presorted sequence without pipeline buffer. Note that RR needs 3-6ms (for buffer size 2) to 3.5-7ms (for buffer size 200) for our test scenes if rendering is turned off. Hence, the CPU time of RR is less than 10% of the total running time.

We also observe that below a certain threshold the number of state changes seems to have no more influence on the running time. We believe that from this point the performance is limited by a different bottleneck. Therefore, according to our experiments a buffer size of 30 is sufficient to achieve almost optimal performance.

Finally, we observe that the speed-up for the PowerPlant scene is smaller than the speed-up for other scenes. The reason is that the PowerPlant scene has no textures. In this scene a state change corresponds to a change of material. Such a change is less expensive than a change of texture. We also remark that in general a change of the shader program is more expensive than the change of a texture. Thus we can expect a better gain of performance in scenes with shader programs.

We conclude that using a pipeline buffer of size 30 with the RR strategy a spatially sorted sequence of primitives can be rendered in almost the same time as if it was sorted by attribute value.

Scene	Town	Aphrodite	Bear	Elephant	PowerPlant
without buffering	145ms	148ms	59ms	60ms	115ms
RR (k=30)	101ms	101ms	42ms	39ms	107ms
presorted	93ms	96ms	39ms	36ms	101ms
speed-up RR	30.3%	31.8%	28.8%	35.0%	7.0%
speed-up presorted	35.9%	35.1%	33.9%	40.0%	12.2%

Table 2: Rendering time of the RR strategy with buffer size 30.

6 Conclusion

We show that our pipeline buffer with the Round-Robin strategy can be used to reduce the number of state changes in the rendering process by an order of magnitude. In comparison to an unsorted sequence our approach reduces the rendering time by up to 35%. Our approach almost achieves the optimal performance of a presorted sequence. It is easy to implement, fast and generic, i.e., it can be used to reduce any type of state change and it can be easily integrated into existing rendering system. It would be interesting to evaluate the performance of our strategy, if it is realized in hardware.

References

[1] T. Aila, V. Miettinen, and P. Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics, 22(3)*, pages 792–800, 2003.

[2] OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley, 1999.

[3] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In *Proc. of ACM SIGGRAPH / EUROGRAPHICS Workshop on Graphics Hardware*, pages 87–95. ACM Press, 2000.

[4] L. Carpenter. The a-buffer, an antialiased hidden surface method. In *Proc. of ACM SIGGRAPH 84*, pages 103–108. ACM Press, 1984.

[5] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[6] S. R. Coorg and S. J. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 83–90, 189. ACM Press, 1997.

[7] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH 93*, pages 231–238, 1993.

[8] P. Lalonde and E. Schenk. Shader-driven compilation of rendering assets. In *Proc. of ACM SIGGRAPH 2002*, pages 713–720. ACM Press, 2002.

[9] Microsoft. *The Microsoft DirectX 9 Programmable Graphics Pipline*. Microsoft Press, 2003.

[10] H. Räcke, C. Sohler, and M. Westermann. Online scheduling for sorting buffers. In *Proc. of the 10th European Symposium on Algorithms (ESA)*, pages 820–832. Springer Verlag, Berlin, 2002.

[11] J. Rohlf and J. Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *Proc. of ACM SIGGRAPH 94*, pages 381–394. ACM Press, 1994.

[12] D. Shreiner. Performance opengl: Platform independent techniques. In *course notes of the ACM SIGGRAPH 2001*, 2001.

[13] C.M. Wittenbrink. R-buffer: A pointerless a-buffer hardware architecture. In *Proc. of ACM SIGGRAPH / EUROGRAPHICS Workshop on Graphics Hardware*, pages 73–80. ACM Press, 2001.

[14] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. In *Proc. of ACM SIGGRAPH 97*, pages 77–88. ACM Press, 1997.