Effective Static Bin Patterns for Sort-Middle Rendering

Bernhard Kerbl
Graz University of Technology
kerbl@icg.tugraz.at

Michael Kenzel
Graz University of Technology
kenzel@icg.tugraz.at

Dieter Schmalstieg
Graz University of Technology
schmalstieg@icg.tugraz.at

Markus Steinberger
Graz University of Technology
steinberger@icg.tugraz.at

ABSTRACT

To effectively utilize an ever increasing number of processors during parallel rendering, hardware and software designers rely on sophisticated load balancing strategies. While dynamic load balancing is a powerful solution, it requires complex work distribution and synchronization mechanisms. Graphics hardware manufacturers have opted to employ static load balancing strategies instead. Specifically, triangle data is distributed to processors based on its overlap with screenspace tiles arranged in a fixed pattern. While the current strategy of using simple patterns for a small number of fast rasterizers achieves formidable performance, it is questionable how this approach will scale as the number of processors increases further. To address this issue, we analyze real-world rendering workloads, derive requirements for effective patterns, and present ten different pattern design strategies based on these requirements. In addition to a theoretical evaluation of these design strategies, we compare the performance of select patterns in a parallel sort-middle software rendering pipeline on an extensive set of triangle data captured from eight recent video games. As a result, we are able to identify a set of patterns that scale well and exhibit significantly improved performance over naïve approaches.

CCS CONCEPTS

•**Computing methodologies** →**Rasterization;** *Massively parallel algorithms;*

KEYWORDS

Static load balancing, Pattern, Parallel Rendering, Sort-middle, GPU

ACM Reference format:
Bernhard Kerbl, Michael Kenzel, Dieter Schmalstieg, and Markus Steinberger. 2017. Effective Static Bin Patterns for Sort-Middle Rendering. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 10 pages.
DOI: 10.1145/3105762.3105777

HPG '17, Los Angeles, CA, USA

978-1-4503-5101-0/17/07…$15.00
DOI: 10.1145/3105762.3105777

1 INTRODUCTION

The key to coping with the massive computational demands of rendering large 3D scenes in real time lies in parallelization. To provide the necessary compute power, a modern graphics processing unit (GPU) features thousands of cores. However, to channel all this power into competitive performance, effective *load balancing* is needed to make sure all cores are best utilized at all times.

In general, load balancing methods can be categorized as being either static or dynamic. Static load balancing distributes the workload following a fixed, predefined scheme solely based on the properties of each individual input element. In contrast, dynamic load balancing can take into account the current state of the GPU which includes, e.g., the load on each individual processor. This increases the scope of possible optimization strategies at the cost of higher conceptual complexity and communication overhead.

Work distribution strategies for parallel rendering have been classified by Molnar et al. (1994) based on where in the pipeline redistribution between processors occurs: before geometry processing (sort-first), between geometry and fragment processing (sort-middle), or after fragment processing (sort-last). In a sort-middle approach, the screen is subdivided into *bins*. Primitives are sorted into the bins they cover. Each bin is then assigned system resources to take on the rasterization of the contents of the bin.

Sort-middle seems to be the preferred strategy on modern devices as the screen coverage of primitives is readily available after geometry processing and the transfer overhead is relatively small before they are split into large numbers of fragments. Rasterization typically uses a coarse-to-fine strategy and is implemented on special-purpose hardware units called *rasterizers*. The assignment of bins to rasterizers commonly follows a fixed pattern. Such a static load balancing pattern seems logical, given that it grants processors exclusive access to the frame buffer in the assigned region, which eliminates the problem of resource contention and has positive effects on cache performance.

Obviously, the design of a binning pattern can have a significant impact on performance. If the distribution of fragments is not sufficiently uniform across screen space, the resulting imbalances will lead to bottlenecks on individual rasterizers. If we reject the option of dynamic load balancing because of its heightened hardware requirements, a good static binning pattern is imperative to achieve high performance on modern processors. Furthermore, as the trend of increasing processor counts continues, the scalability of binning patterns may be of crucial importance for future hardware designs. Thus, we found it surprising that detailed advice on patterns with good load balancing characteristics is scarce in the literature. With

screen resolutions reaching 4K or even 8K and parallel rendering ranging from smart phones to desktop systems and even into the cloud, load balancing strategies become increasingly important.

Static binning patterns must be scalable with respect to bin size, number of rasterizers, and screen size. In this paper, we investigate these issues in detail and make the following contributions:

(1) We determine and analyze the patterns employed on the GPU throughout recent years.
(2) We analyze real-world rendering workloads from recent video games and derive requirements for effective patterns.
(3) We present ten different pattern design strategies based on these requirements and previous work.
(4) We assess and compare the load balancing characteristics of the proposed patterns on more than 200 game scenes.
(5) We evaluate the effects of the most promising patterns on performance in a simulated GPU rendering pipeline.

2 RELATED WORK

In addition to its application in hardware rasterization, subdividing the viewport into spatial bins or tiles has become common practice in the pursuit of high-performance software rendering (Clarberg et al. 2013; Molnar et al. 1992; Patney et al. 2015; Seiler et al. 2008). However, previous approaches were usually not bound by the constraints of a streaming pipeline architecture (i.e., enforcing a fully static bin assignment) and therefore opted for using dynamic load balancing instead (Fuchs et al. 1989; Lin et al. 2001). A notable example for fully-programmable rasterization is presented by Laine and Karras (2011), who demonstrated a complete and parallel software rendering pipeline with tiled rasterization. However, in contrast to a streaming pipeline, their design implies that distribution of clip-space triangle load is known before rasterization, allowing them to perform semi-dynamic load balancing based on bin loads.

Our work assumes a fully-featured streaming pipeline, similar to contemporary hardware rasterization. Thus, we seek to analyze and address impacts and effects of various bin layouts that may be used to target optimal static workload distribution.

2.1 Bin arrangement and size

Previously suggested patterns for subdividing the screen space for parallel processing include using scanlines, horizontal and vertical strips or rectangular tiles (Wang et al. 2011). Juliachs et al. (2007) shuffle 2D portions of the viewport and distribute them to available rasterizers randomly. Eldridge (2001) illustrates a tiling pattern for interleaved tile quads in the renowned *Pomegranate* architecture (Eldridge et al. 2000). However, the authors neither elaborate on how this pattern is produced, nor how its performance would be affected by scaling the tile size or the number of processors.

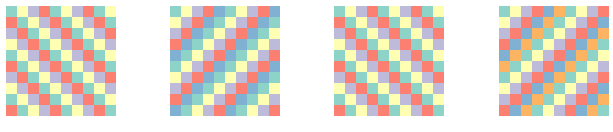
Molnar et al. (1994) recommend using small bin sizes as a means to achieve better load balance. Naturally, as bins get smaller, work is more evenly distributed. However, decreasing bin size also implies an increase of the total workload itself, since triangles have to be processed by every bin they overlap. Thus, the choice of the appropriate bin size is a delicate one and plays an important role in the design of a graphics pipeline (Chen et al. 2005, 1998; Dan Crişu 2012). A mathematical approach to predict performance curves and ideal bin sizes was presented by McManus and Beckmann (1996).

Our work does not address selection of bin sizes, but rather aims to identify guidelines for the arrangement of bins, regardless of their extent, and provide blueprints for effective binning patterns.

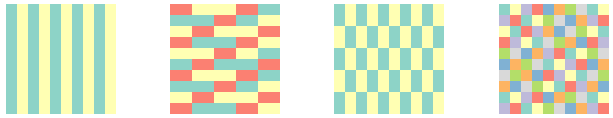
2.2 GPU patterns

On the GPU, physical cores may be grouped together hierarchically to form powerful logical processing units. In recent NVIDIA architectures (NVIDIA 2009), up to 30 streaming multi-processors (SMs), each capable of maintaining thousands of threads, are grouped into a small number of graphics processing clusters (GPCs). Top-level workload distribution for rasterization occurs only on the GPC level of the hierarchy, with each GPC representing one logical rasterizer.

To analyze binning patterns on NVIDIA GPUs, we use a custom GLSL shader and the `NV_shader_thread_group` extension to identify pixel locations that submit to the same group of SMs, and are therefore handled by the same GPC. According to our results, on five out of six recent flag ship models, a diagonal alignment at 45° is used. Bin sizes are consistently small at 16×16 pixels in an effort to mitigate load imbalance (Purcell 2010). We also consider Intel and AMD models, for which we used a timing-based approach to identify screen regions that influence each other's performance in processing fragments. On those architectures, load balancing appears to not be limited to rasterizers alone. For the AMD R9 270X and HD 7870 models, e.g., we detected 8 separate domains distinctly sharing rendered workload, in contrast to the 2 officially documented rasterizers available (AMD 2012). We suspect that this is due to those architectures employing a more fine-grained load balancing concept directly between individual compute units.



(a) GTX 580/680 (b) GTX 780 Ti (c) GTX 1080 (d) Titan Xp



(e) HD 4000 (f) HD 530 (g) 6770M (h) R9 270X

Figure 1: Observed patterns for workload distribution used in GPUs by NVIDIA (a–d), Intel (e, f) and AMD (g, h).

3 GUIDELINES FOR PATTERN DESIGN

An effective pattern design for load balancing should consider the workload characteristics and thus the content being rendered. To this end, we identify fundamental caveats and run statistical tests on versatile rendering content to derive guidelines for pattern design. To obtain a manageable parameter space, we adopt the following restrictions regarding layout combinations for bins and viewports: First, we only consider square bin resolutions, which is common practice in existing software and hardware pipelines. Second, bin sizes are considered to be an immutable property of the underlying architecture. Third, the pattern layout must not depend on high-level parameters, such as window resolution or size. Fourth, all

(a) Age of Mythology (2014) (b) Total War: Shogun 2 (2011) (c) Rise of the Tomb Raider (2015)

Figure 2: Selected game scenes from our data set. Images show the original rendering with an overlay of the output by the software rendering pipeline we used to verify our analytical results. Not shown here: *Deus Ex: Human Revolution* (2011), *Tomb Raider* (2013), *Assassin's Creed IV: Black Flag* (2013), *The Witcher 3: Wild Hunt* (2015) and NVIDIA's *Stone Giant* tech demo.

Age of Mythology capture courtesy of Microsoft. Total War: Shogun 2 capture courtesy of The Creative Assembly. Rise of the Tomb Raider screenshot courtesy of Crystal Dynamics.

analytical and practical evaluations focus on applications running in landscape mode at screen resolutions with a 16:9 aspect ratio.

In the following sections, we assume that a low variance in fragment load across all rasterizers provides a meaningful indicator of effective work balancing and draw conclusions accordingly.

Our guidelines are grounded in both a theoretical analysis and in an analytical evaluation on a representative data set modeling typical GPU workloads. In order to faithfully reproduce realistic GPU rasterization tasks, we have selected seven video games and a recent NVIDIA tech demo (see Figure 2). For each test application, we have captured the triangle stream for at least 20 frame snapshots by injecting a custom DirectX 11 DLL, which saves clip space data directly to a file. All processing was done at 1080p resolution.

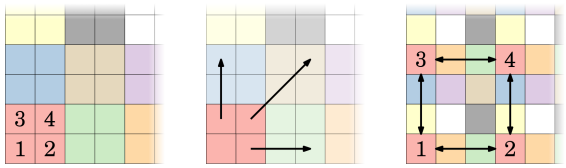
3.1 Space utilization

An efficient binning pattern must consider the number of available rasterizers in its layout. To confirm this statement, consider a naïve binning policy for N parallel rasterizers, where each rasterizer is assigned to an entire row of bins on the viewport (c.f. Crockett and Orloff 1993). Although this policy may achieve satisfactory results with a small bin height h and low number of rasterizers N, choosing either value such that $h \cdot (N-1)$ surpasses the vertical resolution of the viewport implies that at least one rasterizer remains idle. Such a naïve pattern would, therefore, not scale well with increasing rasterizer count. Given the restriction that bin sizes are immutable, the only option towards a scalable binning policy is to pack bins for all available rasterizers into a (preferably small) 2D region and thereby increase the likelihood of achieving high occupancy.

3.2 Local clustering of geometry

In most scenes, geometry is not uniformly distributed (Deering 1993). Decorative elements such as grass, soil or water are represented with low geometric density, while prominent objects, such as trees, edifices or living entities, are designed with a stronger emphasis on geometric detail. These observations suggest that the geometry of a 3D scene, projected to a 2D viewport, has a tendency to form local clusters in screen space. In this case, assigning one rasterizer to contiguous bins could considerably hurt performance.

In order to quantify the influence of local geometry clusters on load balancing, we have assessed the potential improvement that can be achieved by iteratively increasing the distance between bins assigned to the same rasterizer, as outlined in Figure 3. For our evaluation running at 1080p, we first assume a bin size $X \times X$ and subdivide the screen into quad regions containing four bins each (Figure 3a). Each quad is then assigned its own dedicated rasterizer, and we compute the reference standard deviation σ_{ref} over all rasterizers from the ideal fragment load in this configuration. We then proceed to break up the quads by incrementally increasing the distance between the bins assigned to each rasterizer. From each original quad, one bin location is moved two slots to the right, one is moved two slots upward and one is moved both upwards and to the right (Figure 3b). After the first iteration, the gap between bins assigned to the same rasterizer spans two slots (Figure 3c). We continue to space out the bin locations in this way until the distance between the bins for each rasterizer surpasses half of the viewport height. In each iteration i, we record the current standard deviation σ_i of rasterizer workload and compare it to σ_{ref}. The ideal distance between rasterizers is given by the iteration index i that produced the smallest ratio σ/σ_{ref}, multiplied by two.



(a) Initial state (b) First iteration (c) New layout

Figure 3: Progression for spacing out clustered bins.

Figure 4 shows the results of this experiment for several different bin sizes, averaged over our entire data set. Note that the best offset for a bin size $X \times X$ is usually found at or close to distance Δ_X such that $2 \cdot \Delta_X \cdot X = 1080$. Thus, the ideal distance between bins assigned to the same rasterizer is equal to their maximal possible separation. Furthermore, the effect of breaking up the original clusters has a major impact on the variance, reducing σ by at least 30%. Note that, at 160×160, no offset occurred, since $160 \cdot 4 > \frac{1080}{2}$.

3.3 Influence of orientation

Both space utilization and local clustering behavior imply that bins assigned to the same rasterizer should be as widely spaced out as

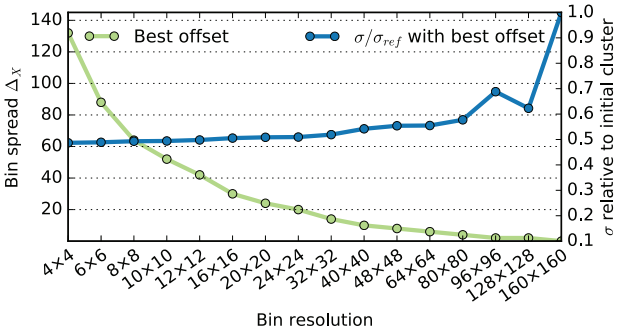
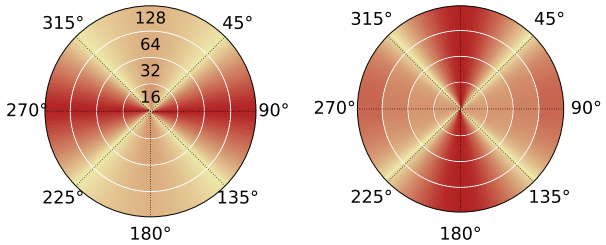


Figure 4: Spreading out bin groups assigned to the same rasterizer lessens impact of triangle clusters and reduces load variance. Smallest σ usually occurs at maximal separation.



(a) Full test set **(b) Age of Mythology**

Figure 5: Variance of fragment load when dividing the viewport into pixel lines with different orientations. Numbers inside the left circle give the line length N that is illustrated by the corresponding ring. Horizontal and vertical directions exhibit high variance (red). Only for *Age of Mythology* is the variance higher with vertical lines than with horizontal.

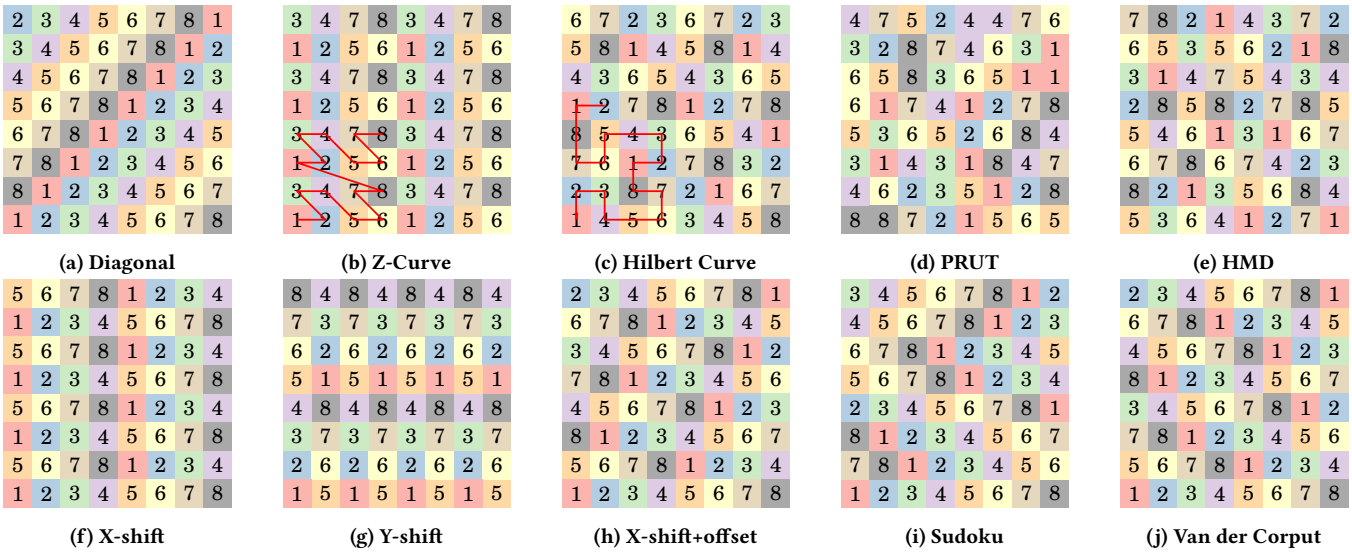
possible in order to avoid local rasterizer repetition. However, we have yet to establish whether the impact of these repetitions is equally severe in all directions. In real-world scenarios, gravity ensures a natural preference of horizontal structures in bedrock, bodies of water and terrain. In contrast, man-made structures, plants and animate entities often stand upright, affording them a superior vantage point. As a matter of fact, very few elements remain that are naturally diagonally aligned. Graphics applications aiming to present realistic scenes exhibit similar structural properties in their geometry. Consequently, we can assume that horizontal and vertical rasterizer repetitions make a pattern more susceptible to localized triangle clusters and therefore more likely to suffer from workload imbalances.

In order to confirm this theory, we analyze the influence on load variance when subdividing the viewport into screen space lines of varying orientation. Samples for computing variance are taken at pixel level by setting up a sliding window of $N \times N$ pixels and sampling N consecutive pixel locations along a line through the center of the sliding window in a given direction, defined by an angular parameter. Line sampling is performed using a Bresenham algorithm aligning with the desired angle. The sum of the fragments submitted to the N pixels in each line is recorded. We then compute the standard deviation σ of the fragment counts for the given direction over all sliding window positions. σ is further normalized for each scene by dividing with the average number of fragments per pixel. This process is performed for a discrete set of directions, yielding the average variance for each tested orientation.

We list the extrema of normalized average standard deviations, as well as the corresponding orientations for all tested applications and sliding window resolutions in Table 1. Furthermore, Figure 5 provides a more intuitive classification of all tested directions on our data set, with each concentric circle representing a different line length N. Based on our analysis, all applications show the highest variance in fragment load between lines oriented at fully horizontal and vertical orientations, thus confirming our initial assumption. With the exception of *Age of Mythology*, horizontal structures appear to have a much bigger impact than vertical ones. This is easily explained: *Age of Mythology* is the only application that enforces a fixed top-down view and has most objects of interest (e.g., units, buildings and resources) vertically aligned. All remaining applications place the viewer at a first or third person perspective. From a view point raised 1-2m above ground, far-reaching planar meshes (e.g., terrain, water, floors, rooftops) tend to line up with the horizon, causing a significant concentration of geometry (and thus overdraw) at horizontal lines. Furthermore, most scenarios place complex objects on or close to a flat surface. Based on these results, we formulate the goal for effective binning to avoid both vertical and horizontal rasterizer repetitions whenever possible.

Table 1: Direction of the lowest and highest average standard deviation (normalized) along all directions for different applications. High σ indicates that an effective pattern should avoid assigning bins along those directions to the same rasterizer.

Samples N (per angle)	AoM		AC 4		Tomb Raider		R.o.t.T.R.		Stone Giant		Shogun 2		Deus Ex: HR		Witcher 3	
	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}	σ_{lo}	σ_{hi}
16	1 (46°)	1.04 (0°)	1 (44°)	1.03 (90°)	0.64 (44°)	0.65 (90°)	0.96 (44°)	0.98 (94°)	0.52 (44°)	0.53 (90°)	1.61 (44°)	1.7 (92°)	0.7 (44°)	0.72 (90°)	1.5 (46°)	1.56 (92°)
32	0.95 (46°)	1 (0°)	0.96 (44°)	1.01 (90°)	0.61 (44°)	0.63 (90°)	0.92 (44°)	0.96 (90°)	0.51 (46°)	0.52 (90°)	1.51 (136°)	1.67 (90°)	0.68 (44°)	0.71 (90°)	1.4 (136°)	1.53 (90°)
64	0.88 (44°)	0.94 (0°)	0.92 (44°)	0.99 (90°)	0.6 (46°)	0.61 (90°)	0.86 (44°)	0.91 (88°)	0.5 (44°)	0.51 (90°)	1.39 (44°)	1.63 (90°)	0.65 (44°)	0.68 (90°)	1.29 (136°)	1.48 (90°)
128	0.78 (136°)	0.85 (0°)	0.84 (44°)	0.95 (90°)	0.56 (44°)	0.59 (90°)	0.78 (44°)	0.85 (88°)	0.47 (44°)	0.5 (90°)	1.22 (44°)	1.58 (90°)	0.60 (44°)	0.66 (90°)	1.15 (46°)	1.42 (90°)



(a) Diagonal (b) Z-Curve (c) Hilbert Curve (d) PRUT (e) HMD

(f) X-shift (g) Y-shift (h) X-shift+offset (i) Sudoku (j) Van der Corput

Figure 6: Illustration of tested rasterizer patterns in bin tiles of size 8. Bins with same color are assigned to the same rasterizer.

4 DESIGNING AND EVALUATING PATTERNS

In this section, we describe ten different patterns based on suggestions from previous work, adaptations of common space filling techniques and our analysis of GPU hardware rasterization. Furthermore, we incorporate the insights gained in Section 3 in an effort to improve existing techniques and design a superior binning policy. We categorize, discuss and compare all patterns based on their basic approach. Finally, we pick the most promising pattern from each category to analyze trends and prospects for their expected effectiveness. All considered patterns (shown in Figure 6) are assessed using realistic clip space geometry from our data set.

Since an ideal binning policy would ensure completely uniform workload among all rasterizers, we rate patterns based on the variance of fragment load they produce for individual rasterizers. We process our input geometry data according to OpenGL conventions and record the amount and distribution of the resulting fragments, generated by the triangles processed in each rasterizer. In order to ensure that the observed trends are generally valid, we always assess variance at multiple bin sizes and rasterizer counts.

As a baseline for comparison, we choose the *Diagonal* pattern (Figure 6a) used in several recent GPU models offered by NVIDIA. This pattern can be generated as follows: Initially, all N rasterizers are lined up in ascending order according to their index. With each row, the index of the first rasterizer is offset by one slot. Indices outside the possible range are wrapped around, creating a repetitive pattern. This policy leads to those bins that are assigned to a given rasterizer forming a diagonal line.

4.1 Space filling curves

Space filling curves are a popular concept for efficiently querying and addressing multidimensional data. In computer graphics, popular applications include creation of spatial data structures, as well as optimizing two-dimensional memory access when programming for the GPU (Nocentino and Rhodes 2010). Here, we assess the performance of two space filling curves that are well-established and routinely employed, namely the *Z-Curve* and the *Hilbert Curve*.

Z-Curve. To produce a *Z-Curve* pattern that covers the entire viewport, we traverse all rasterizer bins and compute the 2D Morton code m_{xy} for each individual bin location, where $(0, 0) \rightarrow 0$ indicates the bin in the lower left corner. The index for the rasterizer to which we assign the bin is then selected as $m_{xy} \bmod N$. Figure 6b shows the corresponding binning pattern, partially overlayed with the Z-Curve.

Hilbert Curve. Similarly to *Z-Curve*, we traverse all bins in the viewport and use the 2D Hilbert distance function $dist_H(x, y)$ to compute the length of the curve at each location, with the bin in the lower left corner specifying the origin. The appropriate rasterizer is chosen from N available indices by calculating $dist_H(x, y) \bmod N$. An examplary layout following the Hilbert Curve in a bin tile of size 8 is shown in Figure 6c.

Evaluation. In Figure 7, we show the rasterizer load variance over our entire data set for *Z-Curve* and *Hilbert Curve*. We plot the recorded values relative to our baseline *Diagonal* at different rasterizer counts. The thick line encompasses results for all tested bin sizes, ranging from 4×4 to 192×192 pixels. The thin opaque line marks the average over all bin sizes. Both patterns appear to perform similarly well; however, *Z-Curve* behaves less consistently with varying bin sizes at low rasterizer counts, as indicated by the significant thickness of the red line. We thus consider *Hilbert Curve* the more suitable pattern and use it as representative for the performance that can be expected from space filling curves.

4.2 Randomized patterns

In computer graphics, randomization is often used as a means to suppress noticeably repetitive artifacts or to create natural-looking shapes and patterns for visual scenes (e.g., Monte Carlo methods).

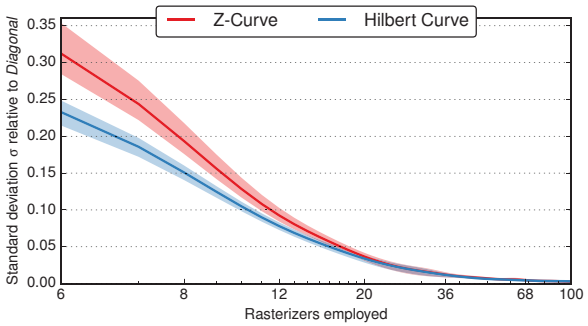


Figure 7: Patterns using space filling curves.

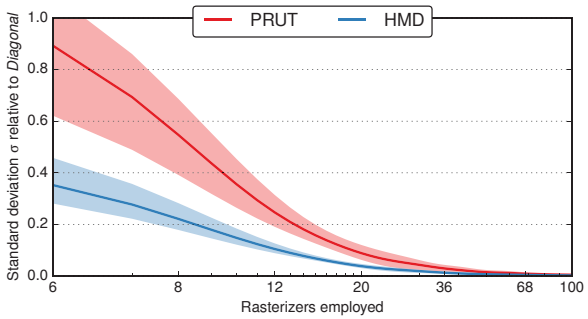


Figure 8: Patterns based on randomization.

To assess randomization for our purposes, we examine two patterns whose layout is entirely defined by randomly generated values.

Pseudo-random uniform distribution. This pattern is generated by traversing all bins over the image domain left-to-right, bottom-to-top and using the Mersenne Twister 19937 with a uniform distribution from 1 to N to choose a rasterizer for each bin at random. Hence, there are no guarantees ensuring minimum separation between bins assigned to the same rasterizer, or even equal occurrence of rasterizers throughout the image domain (Figure 6d).

Note that the space filling curves and the pseudo-random uniform distribution are the only patterns we evaluate that may create a unique, non-repetitive arrangement over the entire viewport. All patterns discussed from here on are defined by periodically repeating bin tiles.

Hierarchical maximized distance. Inside a tile of $N \times N$ bins, we use randomized samples to assign rasterizers in a fashion similar to Poisson disk sampling. To fill all bins in the $N \times N$ tile, we cycle N times through the N available rasterizer indices to ensure equal occurrence of all rasterizers in a tile. At each iteration, we use a simple dart-throwing technique with the rasterizer index as parameter r. We draw up to 50 vacant sample positions inside the $N \times N$ tile at random. In the absence of an ideal value for the Poisson disk radius to test against, we always select the bin that is the farthest from all other bins currently assigned to rasterizer r. The result of this process for a 8×8 tile is illustrated by Figure 6e.

Evaluation. In Figure 8, we show the relative variance against our baseline for *pseudo-random uniform distribution* (PRUT) and *hierarchical maximized distance* (HMD). *HMD* clearly outperforms *PRUT* on all accounts. We attribute the superior effectiveness to the fact that, in contrast to *PRUT*, *HMD* prioritizes large distances between bins assigned to the same rasterizer and thus encourages better space utilization.

4.3 Fixed shift

Instead of shifting bin assignments by a single position in each row, as in the *Diagonal* pattern, we investigate the effect of wider fixed-distance shifts. Based on the trends we identified in the previous section, doing so could benefit significantly from better space utilization and a fairer distribution of localized geometry clusters.

X-shift. A horizontal shift for each row is computed by multiplying the row index i with a fixed value $\Delta = \frac{N}{k}$, where N is the number of rasterizers, and k gives the number of rows until the pattern repeats. The corresponding shift in row i can thus be computed as $\Delta_i = \lfloor \frac{i \cdot N}{k} \rfloor \bmod N$. Hence, *X-shift* forms rectangular, but not necessarily square, periodically repeating $N \times k$ tiles (see Figure 6f). Notice that this implies a vertical rasterizer repetition every k rows. In order to maximize the Euclidean distance between any two bins assigned to the same rasterizer, we choose k close to the square root of N as $\lfloor \sqrt{N} \rfloor$.

Y-shift. Y-shift is essentially a rotated version of *X-shift*. Shift parameters are chosen identically. However, instead of a horizontal shift per row, a vertical shift is applied per column (see Figure 6g). Similarly to *X-shift*, the *Y-shift* pattern repeats every k columns and, thus, has a tendency towards horizontal rasterizer repetitions.

X-shift+offset. While *Diagonal* has a small minimum distance between bins assigned to the same rasterizer, *X-shift* has a higher rate of rasterizer repetition on the vertical axis, making it a potentially weak candidate in rendering scenarios with prominent vertical structures (e.g., *Age of Mythology*). *X-shift+offset* aims to combine the benefits of both approaches: based on the shift function for *X-shift*, row arrangements are offset by one position every k rows. The shift in row i can thus be computed as $\Delta_i = \lfloor \frac{i \cdot (N+1)}{k} \rfloor \bmod N$. Though this modification may be minor, it effectively ensures that vertical rasterizer repetitions do not occur before passing N rows (see Figure 6h). Thus, instead of periodic $N \times k$ tiles where $k \ll N$, *X-shift+offset* can fill a full $N \times N$ tile of bins before repeating itself.

Evaluation. In Figure 9, we compare fixed shift patterns *X-shift*, *Y-shift* and *X-shift+offset* relative to *Diagonal*. When applied to our full data set, *Y-shift* is clearly trailing behind the other alternatives. This comes as no surprise: According to the directional analysis of our data set, horizontally aligned geometry is much more prominent; thus, the frequent horizontal rasterizer repetitions in *Y-shift* cause its performance to falter. A clear exception to this observed trend is posed by *Age of Mythology*, for which the performance of *Y-shift* is generally on par with, and, in isolated cases, clearly better than *X-shift*. However, both patterns are bested on all occasions by *X-shift+offset*, which we ascribe to the fact that *X-shift+offset* effectively reduces both horizontal and vertical rasterizer repetitions.

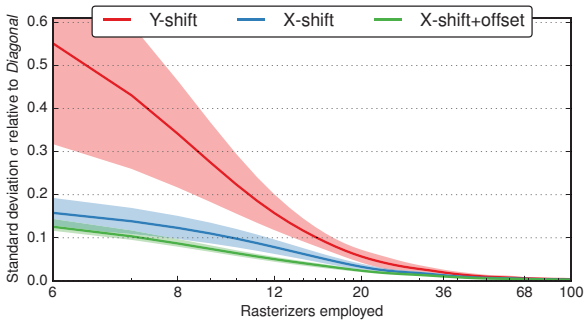


Figure 9: Comparing fixed shift patterns.

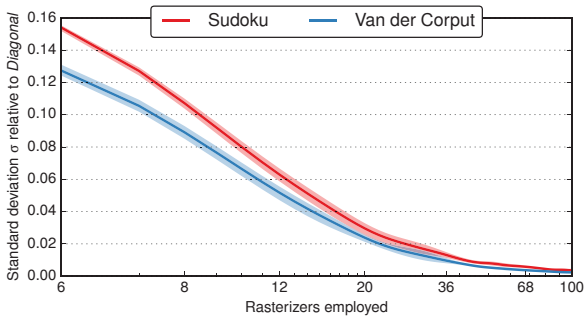


Figure 10: Comparing variable shift patterns.

4.4 Variable shift

In addition to patterns obtained by applying a fixed shift, we consider two instances of shift patterns with less predictable behavior.

Sudoku. As part of their discussion on suitable binning patterns, Eldridge (2001) implicitly suggest that uniform workload distribution over periodically repeating $N \times N$ tiles can be facilitated by requiring that no two bins in the same row or column are assigned to the same rasterizer. Due to its similarity, we call this strategy *Sudoku*, after the popular Japanese puzzle. A random $N \times N$ tile that fulfills the *Sudoku* constraint can be quickly generated by drawing a random shifting value for each row in the interval $[0, N)$ and disallowing choosing the same shift value twice. Figure 6i shows one arrangement following the Sudoku policy in a bin tile of size 8.

Van der Corput. For this shift-based pattern, row shifts are computed based on a base 2 Van der Corput low-discrepancy sequence $\left(0, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \dots\right)$. The shift in row i is given by the i^{th} element in the sequence, multiplied by the next higher power of two for the number of rasterizers, $2^{\lceil \text{ld} N \rceil}$. All shifts greater than N are skipped, generating a non-repeating pattern inside a bin tile of size N. Note that this pattern also implicitly fulfills the constraint that *Sudoku* is based on. Figure 6j shows the definite arrangement of a bin tile that is generated by this method for $N = 8$.

Evaluation. Figure 10 compares the relative variance for *Sudoku* and *Van der Corput*. Both patterns show very similar development for varying rasterizer counts and bin sizes. We attribute this circumstance to their shared quality, namely the constraint of allowing rasterizers only once per row and column. Although both patterns appear to be very effective at distributing workload evenly, *Van der Corput* exhibits an evident advantage over *Sudoku*.

4.5 Comparison of all categories

Finally, we compare the most promising patterns from all categories and assess their characteristics and overall behavior. Figure 11 shows the development of all approaches at three different bin sizes. In order to stress the possible benefits of using one pattern over another, we plot the coefficient of the variation $c_v = \frac{\sigma}{\mu}$. The choice of using c_v over σ is motivated by the fact that, in contrast to comparing quality of workload distribution, the standard deviation cannot adequately quantify the exact potential for improvement without knowing the average load per rasterizer μ.

Trends for low rasterizer counts are identical at all evaluated bin resolutions: *Diagonal* quickly falls behind all other techniques due to its poor handling of clusters and space utilization. All other patterns exhibit a much slower growth of c_v, with *X-shift+offset* and *Van der Corput* tied for best performance. However, for higher bin resolutions and rasterizers counts, *X-shift+offset* gradually falls behind. For bigger bin sizes, the differences between the techniques (with the exception of *Diagonal*) become less pronounced. Note the ranges of the plotted coefficients for the respective bin sizes. The recorded values of c_v at 128×128 differ from those at 16×16 by more than one order of magnitude. Thus, differences between the individual techniques have a much higher relative impact at bigger bin sizes in terms of performance. The most promising pattern out of those evaluated is *Van der Corput*, with its coefficient of variation consistently below or on par with its contenders.

Considering our previously stated guidelines for pattern design, the high performance of *Van der Corput* is not surprising. In each 2D bin tile ($N \times N$), each rasterizer is referenced with the same frequency, which leads to good space utilization. Within each row, the distance between bins assigned to the same rasterizer is maximal. The same is true for each individual column. Thus, horizontally and vertically dense regions will be assigned to the same rasterizer only when there is no way to avoid that. Moreover, the pattern generation rule ensures that the 2D distance between rasterizers is always high, avoiding local clusters. The direction in which rasterizer assignments repeat within a 2D region is loosely oriented along a 45° angle. All these considerations also apply for *X-shift+offset*, which also shows a very competitive performance. However, *Van der Corput* is locally less structured than *X-shift+offset*, which probably is the reason for giving it an additional edge over the other approaches, especially when the rasterizer count increases.

4.6 Influence of Partitioning

So far, we have evaluated patterns with typical GPU workloads for complete frames. However, when rendering large scenes, the GPU cannot process all primitives at once and only a limited number of primitives are in flight concurrently. In order to understand the influence of this workload partitioning in the context of binning patterns, we split the input triangle stream of each scene into M batches of equal size, while maintaining the order of primitives as they were submitted to the GPU. We then simulate the binning of each individual batch separately, evaluate the resulting temporally

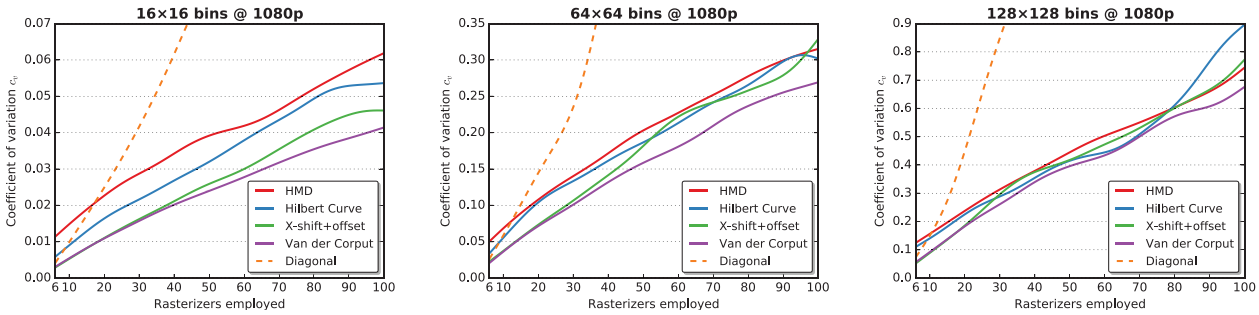


Figure 11: Performance comparison on our data set for the most promising patterns, alongside the *Diagonal* baseline. The coefficient of variation c_v estimates the imbalance in rasterizer workload for each pattern at different configurations. While *X-shift+offset* performs as well as *Van der Corput* for low rasterizer counts, they usually start to diverge at ~20 rasterizers.

local variance, and compute an average c_v from all M individual batches. By performing this process with several different numbers of partitions M, we can identify trends that are likely to influence pattern performance on actual hardware.

Figure 12a shows the influence of the batch size on c_v when using 6 rasterizers and a bin size of 16×16. This is equivalent to the configuration used in the NVIDIA GeForce Titan Xp and thus representative of current graphics hardware. For very small batch sizes, the relative variance is high, as only few triangles are rendered and few bins actually receive workload. However, as the batch size increases to $10 - 25\%$ of the full scene, c_v already converges to the figures previously measured for the entire frame. This points towards patterns already performing as expected when only a small portion of the scene can be processed in parallel. As the amount of rendered geometric detail is steadily increasing, this result also points towards the necessity to be able to process at least 10% of a scene at once to achieve close-to-ideal load balancing.

We further consider the expected benefit of choosing one pattern over another for these smaller workloads. To visualize their relative behavior in detail, we plot the development of c_v as the degree of partitioning increases, relative to *Van der Corput* as a reference in Figure 12b. The relative difference in performance between patterns steadily decreases as the batch size becomes smaller. However, when using batches the size of $1/100$ of the full scene, the difference between the best and worst pattern still makes for a factor of 2×. Like in our previous experiments, both *X-shift+offset* and *Van der Corput* remain the most efficient methods, even though the initial advantage of *X-shift+offset* degrades with batches getting smaller. Surprisingly, we found that *HMD* approaches more advanced patterns with increasing degree of partitioning, and even reaches sophisticated space filling curves. This can be explained by the fact that smaller batches affect a limited portion of the screen and *HMD* explicitly considers separation of rasterizers in its layout, which is the most dominant factor when rendering few primitives.

As can be seen in Figure 12c, increasing the number of rasterizers to 20 results in a stronger influence of partitioning on *Diagonal*. While *Diagonal* performs equally well as *HMD* when processing the entire scene at once, it fails to show relative improvement and is eventually even overtaken by the straightforward *PRUT* pattern. Thus, as the size of batches decreases, *Diagonal* eventually falls behind all other alternatives. Furthermore, we find this setup to be one of several instances where *Z-Curve* eventually beats *Hilbert Curve*. This is not a general rule, but still a common occurrence, which solidifies our initial impression that space filling curves should be considered on a case-by-case basis, since there is no single best choice for all configurations.

Finally, we found that partitioning favorably affects *X-shift+offset* when using bigger bin sizes with high rasterizer counts. In such configurations, as shown above, the pattern usually starts to trail behind *Van der Corput* when processing entire scenes. However, increasing the number of batches the scene is partitioned into causes *X-shift+offset* to quickly approach the reference. We consider this circumstance an argument for the general usability of *X-shift+offset*, as its divergence from *Van der Corput* appears to be mitigated by the (likely) partitioning of large scene input data.

4.7 Discussion

The coefficient of variation (see Figure 11) lets us speculate on how the employed patterns may translate into performance on real hardware. For a bin size of 16×16 @ 1080p, where a bin covers $1/120$ of the screen horizontally, up to 18 rasterizers achieve a $c_v < 1\%$ when rendering entire frames using the best pattern. This setup corresponds to the bin size employed on current NVIDIA GPU designs, which often rely on a diagonal pattern. Translated into runtime performance, rasterizers must handle a load imbalance of less than 1% on average. Considering that there might be other system-wide load balancing strategies happening concurrently, for instance, with vertex processing on the GPU, one would probably not expect any measurable performance hit. If a diagonal pattern is used, the 1% threshold is already exceeded with 10 rasterizers.

We hypothesize that the challenges of finding a static pattern that ensures equally uniform load balancing across many processors is one reason for using relatively few logical rasterizers (GPCs) in current GPU designs, even as the SM count is increased. A consequence of this design choice is that more advanced dynamic load balancing strategies between a rasterizer and its associated multiprocessors are needed.

Naturally, the demands on the rasterizer itself also increase as the resolution increases. Since the graphics pipeline must enforce primitive order during rasterization (Purcell 2010), the rasterizer is not completely free in its strategies for parallelization. Thus, it is questionable whether rasterization performance can be scaled

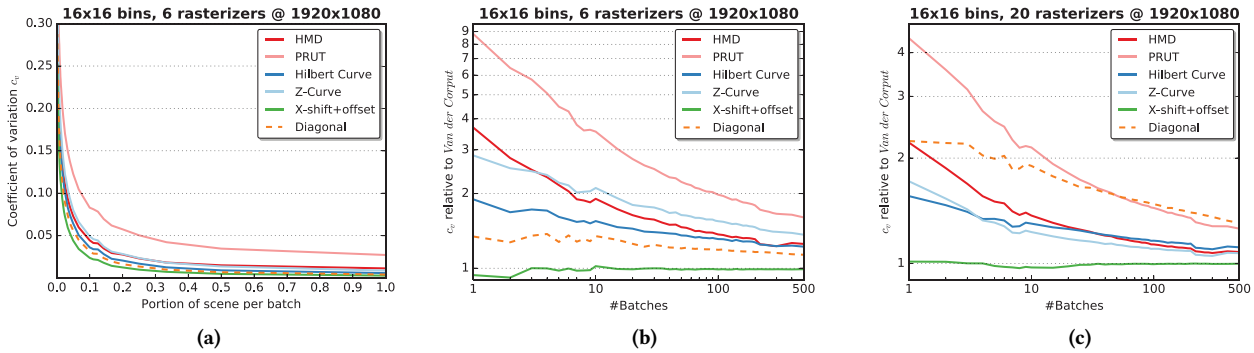


Figure 12: Developments in pattern performance with regard to scene partitioning. (a) For configurations corresponding to current hardware, partitioning strongly affects c_v and thus the expected quality of load balancing. Measurements converge toward previous results at a batch size of about 1/10 of the scene. (b) Relative to *Van der Corput*, *HMD* converges toward similar performance as space filling curves. As with increasing the amount of rasterizers, partitioning causes *Diagonal* to eventually deteriorate. Configuration (c) shows *Diagonal* being overtaken by the trivial *PRUT*, and *Z-Curve* outperforming *Hilbert Curve*.

along with fragment processing performance in future hardware designs without increasing the number of rasterizers.

With larger bin sizes, the influence of the pattern becomes more severe. In these cases, our best contender, the *Van der Corput* pattern, deviates from an ideal distribution by up to 10% when rendering full frames with 30 rasterizers and a bin size of 1/30 of the viewport width, and 25% with a bin size of 1/15. Such larger bin sizes relative to the viewport width might be found on mobile devices, which render in lower resolution, or in software-based rendering, which tries to avoid communication overhead.

Analyzing the influence of workload partitioning on rasterizer load variance, our experiments indicate that about $10 - 20\%$ of the scene should be processed in parallel. In this case, we see an equally good work distribution as if the entire scene was processed as one. When processing only small geometry batches, the relative performance of the patterns hardly change, indicating that our pattern design criteria are largely independent of the amount of data being rendered. This is not surprising, as most considerations guide local bin assignment rather than a global strategy.

It is tempting to draw the conclusion that a small bin size is essential for good performance. However, our analysis only considers the load balancing characteristics and ignores the cost for data transfer and duplication. A smaller bin size will quickly lead to increased communication overhead, as the same triangle has to be transmitted to multiple rasterizers. Depending on implementation, this overhead may outweigh the benefits of smaller bin sizes.

5 PARALLEL SOFTWARE SIMULATION

In order to test our pattern designs in a complete system and to verify theoretical results for their impact on rendering performance, we have built a software rendering pipeline running in CUDA, which supports arbitrary patterns for rasterization (Kenzel et al. 2017). The renderer follows a full streaming model, employs a sort-middle design, and performs dynamic load balancing between geometry processing and rasterization/fragment processing. Furthermore, it does not process the entire scene at once, but gradually streams the data to the rasterizers, overall mimicking the GPU pipeline. Thus, this experimental setup gives an accurate estimate of the performance influence of the tested patterns. We have extended the original input geometry with auxiliary data listing the appropriate rasterizer indices for every triangle. We precompute these data for all test scenes, bin sizes and patterns and load them during rendering to perform load balancing according to the tested pattern.

We process our test data set at two different bin sizes of 16×16 and 64×64 pixels and assess pattern performance for rendering at 1080p resolution. As we do not have access to the hardware rasterizers (GPCs) directly, our implementation employs an equal number of rasterizers on each SM in software. Thus, the rasterizer count must be either smaller than or a multiple of the SM count, and we run our tests using 6, 20 and 60 logical rasterizers. We simulate elaborate fragment shading by performing 2500 fused multiply-add (FMA) instructions before submitting the fragment color.

Table 2 shows the average achieved frame rates as frames per second (FPS) for running the full test set with each technique at different bin sizes and rasterizer counts. The numbers in brackets further state the harmonic mean of the average relative speed-up in each scene over the baseline set by *Diagonal*. The behavior of the patterns at different settings closely matches our predictions: with higher numbers of rasterizers and bigger bin size, the impact of choosing a sophisticated pattern increases, outclassing *Diagonal* by a factor of almost 1.9× using 60 rasterizers at 1080p. The relative performance gains between different patterns is also amplified with the rise of either parameter. Overall, we can summarize that both *X-shift+offset* and *Van der Corput* perform the strongest, with the former dominating at low and the latter at high rasterizer counts.

6 CONCLUSION

We have identified and analyzed possible influential factors on performance to be considered when designing a static binning pattern for sort-middle rasterization. In an effort to optimize load balancing behavior, we have presented several different examples of patterns with distinct characteristics and assessed them both analytically and practically. Runtime measurements for each pattern running at various configurations were obtained using an efficient software

Table 2: Results for tested patterns on our data set at multiple configurations. For each pattern, we show average achieved frame rate, as well as relative speed-up over *Diagonal*. The technique with best performance is marked bold for every setup.

	FPS (speedup) with 16×16 bins				FPS (speedup) with 64×64 bins			
#Rasterizers	Hilbert	HMD	X-shift+offset	Van der Corput	Hilbert	HMD	X-shift+offset	Van der Corput
6	3.3 (1.00)	3.3 (1.00)	**3.3 (1.00)**	3.3 (1.00)	3.3 (0.99)	3.3 (0.99)	**3.4 (1.01)**	3.4 (1.01)
20	10.9 (1.00)	10.9 (1.00)	**11 (1.01)**	11 (1.01)	9.5 (1.02)	9.6 (1.04)	**10.3 (1.12)**	10.3 (1.11)
60	16.2 (1.09)	16.1 (1.08)	16.4 (1.10)	**16.4 (1.10)**	10.6 (1.72)	10.3 (1.70)	11.0 (1.78)	**11.6 (1.89)**

rendering pipeline on the GPU. Based on our predictions and their confirmation from the measured runtime results, we have successfully identified a set of patterns that scale well with the number of rasterizers and exhibit significantly improved performance over naïve approaches. Specifically, we have identified two deterministic patterns that exhibit close-to-ideal behavior and are easy to apply. Although performance gains may be negligible for small bin sizes and low rasterizer counts, they become more pronounced when either of these values grows. The desire for high resolutions and the communication penalty of small bin sizes implies the need for more rasterizers, in turn creating a strong need for a good binning pattern in future hardware designs.

ACKNOWLEDGMENTS

This research was supported by the DFG grant STE 2565/1-1 and the Austrian Science Fund (FWF) I 3007.

REFERENCES

AMD. 2012. White Paper: AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf. (2012). Retrieved June 16, 2017.

Jiawen Chen, Michael I. Gordon, William Thies, Matthias Zwicker, Kari Pulli, and Frédo Durand. 2005. A Reconfigurable Architecture for Load-balanced Rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '05)*. ACM, New York, NY, USA, 71–80. DOI: http://dx.doi.org/10.1145/1071866.1071878

Milton Chen, Gordon Stall, Homan Igehy, Kekoa Proudfoot, and Pat Hanrahan. 1998. Simple Models of the Impact of Overlap in Bucket Rendering. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, S. N. Spencer (Ed.). The Eurographics Association. DOI: http://dx.doi.org/10.2312/EGGH/EGGH98/105-112

Petrik Clarberg, Robert Toth, and Jacob Munkberg. 2013. A Sort-based Deferred Shading Architecture for Decoupled Sampling. *ACM Trans. Graph.* 32, 4, Article 141 (July 2013), 10 pages. DOI: http://dx.doi.org/10.1145/2461912.2462022

Thomas W. Crockett and Tobias Orloff. 1993. A MIMD Rendering Algorithm for Distributed Memory Architectures. In *Proceedings of the 1993 Symposium on Parallel Rendering (PRS '93)*. ACM, New York, NY, USA, 35–42. DOI: http://dx.doi.org/10.1145/166181.166186

Dan Crişu. 2012. *Hardware algorithms for tile-based real-time rendering.* Ph.D. Dissertation. Delft University of Technology.

M. F. Deering. 1993. Data complexity for virtual reality: where do all the triangles go?. In *Proceedings of IEEE Virtual Reality Annual International Symposium*. 357 – 363.

Matthew Eldridge, Homan Igehy, and Pat Hanrahan. 2000. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 443–454. DOI: http://dx.doi.org/10.1145/344779.344981

Matthew Willard Eldridge. 2001. *Designing Graphics Architectures Around Scalability and Communication.* Ph.D. Dissertation. Advisor(s) Hanrahan, Pat. AAI3026802.

Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. 1989. Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-enhanced Memories. *SIGGRAPH Comput. Graph.* 23, 3 (July 1989), 79–88. DOI: http://dx.doi.org/10.1145/74334.74341

M. Juliachs, T. Carrard, and J.-P. Nominé. 2007. Hybrid CPU-GPU Unstructured Meshes Parallel Volume Rendering on PC Clusters. In *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization (EGPGV '07)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 85–92. DOI: http://dx.doi.org/10.2312/EGPGV/EGPGV07/085-092

Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2017. A High-Performance Software Graphics Pipeline Architecture for the GPU. (2017). to appear.

Samuli Laine and Tero Karras. 2011. High-performance Software Rasterization on GPUs. In *Proc. High Performance Graphics (HPG '11)*. 79–88.

Wai-Sum Lin, Rynson W. H. Lau, Kai Hwang, Xiaola Lin, and Paul Y. S. Cheung. 2001. Adaptive Parallel Rendering on Multiprocessors and Workstation Clusters. *IEEE Trans. Parallel Distrib. Syst.* 12, 3 (March 2001), 241–258. DOI: http://dx.doi.org/10.1109/71.914755

Donald McManus and Carl Beckmann. 1996. Optimal Static 2-dimensional Screen Subdivision for Parallel Rasterization Architectures. In *Proceedings of the Eleventh Eurographics Conference on Graphics Hardware (EGGH'96)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 59–67. DOI: http://dx.doi.org/10.2312/EGGH/EGGH96/059-067

Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. 1994. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.* 14, 4 (July 1994), 23–32. DOI: http://dx.doi.org/10.1109/38.291528

Steven Molnar, John Eyles, and John Poulton. 1992. PixelFlow: High-speed Rendering Using Image Composition. *SIGGRAPH Comput. Graph.* 26, 2 (July 1992), 231–240. DOI: http://dx.doi.org/10.1145/142920.134067

Anthony E. Nocentino and Philip J. Rhodes. 2010. Optimizing Memory Access on GPUs Using Morton Order Indexing. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10)*. ACM, New York, NY, USA, Article 18, 4 pages. DOI: http://dx.doi.org/10.1145/1900008.1900035

NVIDIA. 2009. Whitepaper: NVIDIAfis Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. (2009). Retrieved April 6, 2017.

Anjul Patney, Stanley Tzeng, Kerry A. Seitz, Jr., and John D. Owens. 2015. Piko: A Framework for Authoring Programmable Graphics Pipelines. *ACM Trans. Graph.* 34, 4, Article 147 (July 2015), 13 pages. DOI: http://dx.doi.org/10.1145/2766973

Tim Purcell. 2010. Fast Tessellated Rendering on the Fermi GF100. In *High Performance Graphics Conf., Hot 3D presentation.*

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Trans. Graph.* 27, 3, Article 18 (Aug. 2008), 15 pages. DOI: http://dx.doi.org/10.1145/1360612.1360617

Lizhe Wang, Dan Chen, Ze Deng, and Fang Huang. 2011. Review: Large Scale Distributed Visualization on Computational Grids: A Review. *Comput. Electr. Eng.* 37, 4 (July 2011), 403–416. DOI: http://dx.doi.org/10.1016/j.compeleceng.2011.05.010