Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

Hexagonal Storage Scheme for Interleaved Frame Buffers and Textures

Yosuke Bando Takahiro Saito Masahiro Fujita

TOSHIBA Corporation
{yosuke1.bando, takahiro4.saito, masahiro1.fujita}@toshiba.co.jp

Abstract

This paper presents a storage scheme which statically assigns pixel/texel coordinates to multiple memory banks in order to minimize frame buffer and texture memory access load imbalance. In this scheme, the pixels stored in a particular memory bank are placed at the center and the vertices of hexagons packed in the frame buffer. By making these hexagons close to regular so that the pixel placement is uniform and isotropic, frame buffer and texture memory accesses are evenly distributed over the memory banks. The analysis of memory access patterns in rendering typical 3D graphics scenes shows that the hexagonal storage scheme can reduce rendering performance degradation due to bank conflicts by an average of 10% compared to the traditional rectangular storage scheme.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Parallel Processing

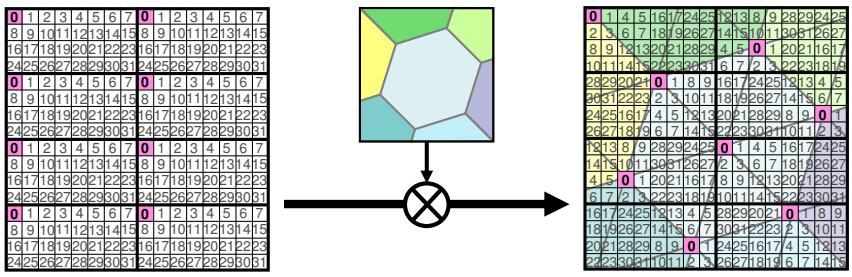


Figure 1: *The presented storage scheme assigns pixel coordinates to multiple (in this case 32) memory banks, by permuting the bank IDs in each rectangle in a traditional rectangularly interleaved bank ID assignments (left), so that the pixels having the same bank ID are placed at the center and the vertices of nearly regular hexagons packed in the frame buffer (right).*

1. Introduction

Graphics processing units (GPU) have been endowed with increasing parallelism in order to meet the demand for greater graphics acceleration. Accordingly, the amounts of pixel and texel data transfer per unit time have also been increasing due to parallel updates of the frame buffer and parallel texture fetches. Typically, this high memory bandwidth requirement can be alleviated using multiple memory banks, each of which stores disjoint portions of the frame buffer or texture so that the multiple portions of the memory can be accessed in parallel. However, this configuration incurs memory access load imbalance because each pixel is stored in a fixed memory bank depending on its coordinate on the 2D rectangular frame buffer, and therefore memory accesses cannot be dynamically distributed to the memory banks based on temporal load.

This paper presents a storage scheme which statically assigns pixel coordinates to memory banks in order to minimize memory access load imbalance resulting from one-to-one correspondence between coordinates and banks. The strategy is to distribute the pixels stored in a particular memory bank uniformly and isotropically on the frame buffer. The ideal placement is to locate the pixels at the center and the vertices of regular hexagons packed in the frame buffer.

However, it is impossible to form regular hexagons by choosing their vertices from a square grid of pixels. A triangular or hexagonal grid of pixels [Gla94, Tyt00] may not be subject to this problem, but GPUs with such shapes of pixels have yet to become readily available commercially. Therefore, we present our approach to finding an assignment of square grid pixel coordinates to the memory banks in which the pixels assigned to the same memory bank form nearly regular hexagons. And we show the obtained assignments for several variations of the number of memory banks. By analyzing memory access patterns in rendering several typical 3D graphics scenes, we demonstrate that our storage scheme distributes both frame buffer and texture memory accesses over the memory banks more evenly than do other non-hexagonal storage schemes.

2. Related Work

We focus on static storage schemes and do not cover adaptive or dynamic memory assignments, such as the MAHD algorithms [Mue95] and the demand-paging algorithm used in the VC-1 [NK96].

Static assignments of regions of the frame buffers to memory banks are often used in conjunction with *sort-first* and *sort-middle* architectures defined in [MCEF94], where each graphics processor is responsible for the regions of the frame buffer stored in the associated memory bank as shown in Figure 2(a). This configuration allows simple connections between the graphics processors and the memory, but load balancing of multiple graphics processors is directly affected by the storage scheme in use.

In this context, two typical storage schemes are described in [FvDFH90]: the one involves dividing the frame buffer into N rectangular regions, and assigning each region to one of the N memory banks; the other involves dividing the frame buffer into many small regions and distributing them to the memory banks in a finely interleaved manner. As tessellations of 3D objects are getting finer [Dee93], the latter storage scheme becomes preferable since it prevents primitives from being drawn to one region.

In most of the existing static and finely interleaved patterns found in literature [Fuc77, Par80, PH89, FvDFH90, Mue95], the distribution of the regions assigned to the same memory bank is aligned to the horizontal and vertical axes of the frame buffer. One such example for $N = 8$ is shown in the left hand side of Figure 7. However, this rectangular distribution somewhat lacks uniformity and isotropy [Gla94], which can increase chances of bank conflicts.

The storage scheme used in *Multiaccess Frame Buffer (MFB)* proposed by Harper [Har94] is one of a few examples of non-axis-aligned bank assignments. In MFB scheme, pixels in any rectangle whose area is $N/2$ are assigned to mutually different memory banks (however, it was shown that this property is not always satisfied [Wei96]). Although the major intention of this assignment is to permit parallel frame buffer updates in units of constant area rectangles, this storage scheme can be viewed as a variation of finely interleaved patterns for memory access load balancing. However, no analysis of the memory access patterns based on this scheme in rendering typical 3D graphics scenes is presented. This paper analyzes the memory access patterns resulting from various storage schemes including typical axis-aligned assignments, MFB scheme, and our hexagonal one, and evaluates their differences.

Our storage scheme is applicable to frame buffers in a *sort-last* architecture [MCEF94] shown in Figure 2(b) and to textures in a *shared texture memory* architecture [IEH99] shown in Figure 2(c) as long as pixels/texels are stored in the memory banks in an interleaved manner as described above. We show results for both frame buffers and textures.

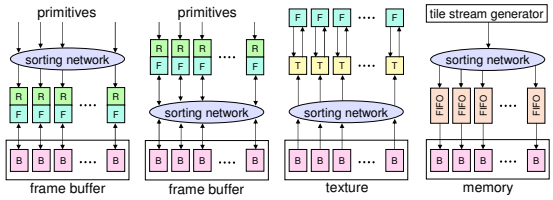


Figure 2: *Graphics subsystems with multiple memory banks. R: rasterizer, F: fragment processor, T: texture unit, and B: memory bank. (a) Sort-middle architecture where each F is directly linked to one of the B's. (b) Sort-last architecture with a pixel sorting network between the F's and the B's. (c) Shared texture memory architecture with a texel sorting network between the T's and the B's. (d) A model for evaluating performance of storage schemes.*

3. Hexagonal Storage Scheme

The following explanation is for the case of frame buffers, but it can be directly applied to the case of textures by substituting "texture" for "frame buffer," and "texel" for "pixel."

3.1. Preliminaries

We consider a frame buffer memory system with N memory banks. We assume that N is a power of two. Each memory bank has a serial number called *bank ID* $\in \{0, 1, \cdots, N-1\}$, and the frame buffer is divided into a number of small disjoint square regions called *tiles*, each of which is assigned a fixed bank ID. The tile size can be equal to the pixel size, but pixels are often clustered into tiles for several reasons: e.g., alignment with the units of data transfer or with the cache line size; LOD calculation using adjacent fragments; SIMD operations on fragments; fast clear of buffers; and tile-based compression.

To simplify the calculation of a tile's storage location

within a memory bank, we cluster N tiles into a rectangle called *block*. These N tiles have mutually different bank IDs. In this configuration, a serial number for a block called *block ID* corresponds to the location within each of the memory banks where the tile in that block is stored. Block IDs can be assigned in row-major order as shown in Figure 3. Alternatively, block IDs can be swizzled [AMH02] if it is acceptable to confine the buffer size to a power of two times the block size, which is often the case with textures. An example of swizzling applied to the pixel order is shown in Figure 3.

We consider assigning a memory bank to each tile so that the tiles stored in a particular memory bank are distributed uniformly and isotropically over the frame buffer, since this minimizes the chance that each bank is accessed multiple times in a short period of time. Such distribution is achieved by placing the tiles at the center and the vertices of regular hexagons packed in the frame buffer as shown in Figure 4(a), which maximizes the distances between the neighboring tiles [CS98]. However, as Figure 4(b) shows, the tiles would not fit in the grid since the positions of the tiles do not fall on integer locations. Section 3.2 describes how we find an assignment of bank IDs to the grid of tiles by which the tiles labeled the same bank ID form nearly regular hexagons.

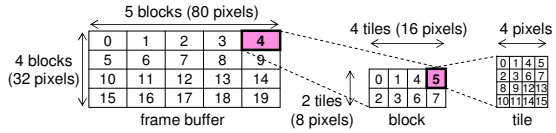
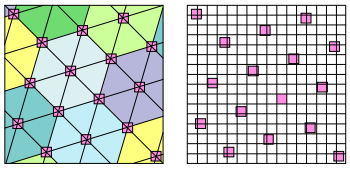


Figure 3: *Example of blocks and tiles. A frame buffer is divided into blocks, each of which is assigned a block ID. A block is divided into tiles, each of which is assigned a bank ID. The pixels in a tile in this figure are in a swizzled order.*



(a) (b)

Figure 4: *Example of regular-hexagonal placement of tiles. 16 tiles are stored in one memory bank for $N = 16$ with a buffer size of 16×16 tiles. (a) Tiles (shown as small squares) are placed at the vertices of regular hexagons. (b) The tiles at the same location as in (a) do not fit in a square grid.*

3.2. Search for Hexagonal Bank ID Assignment

Currently, we obtain the desired bank ID assignments by a brute-force search. We evaluate uniformity of an assignment by the side lengths of the Delaunay triangles formed by the tiles having the same ID. We adopt the assignment with the largest minimum side length. If there are multiple such assignments, we choose the assignment with the smallest average side length because it has sides of similar length and is therefore isotropic.

An exhaustive search of possible bank ID assignments is impractical due to combinatorial explosion: there are $(N!)^{k-1}$ combinations of assignments where k is the number of blocks. The reason for -1 in the exponent is because we can fix the assignment in one block in order to eliminate the redundant $N!$ assignments resulting from bank ID permutation. Unfortunately, we could not figure out whether a polynomial-time algorithm exists for this optimization problem. Therefore, we introduced the following three constraints to reduce the search space.

First, we search recursively based on the solution for a smaller N. Given the assignment for N, we double its tile size, and we consider placing four original size tiles labeled $4i, 4i+1, 4i+2$, and $4i+3$ in the double size tile labeled $i (\in \{0, 1, \cdots, N-1\})$ in each block to obtain assignments for $4N$ (see Figure 5). This reduces the number of combinations to $(4!)^{N(k-1)}$. The base cases ($N = 1, 2$) for this recursion are given in Figure 6. Since the block size is also doubled at every step of the recursion, the block will always be a square for an even $n (= \log_2 N)$, and half of a square for an odd n.

Second, we only accept assignments that are equitable to all memory banks: i.e., the pattern of the tiles with a particular bank ID on the frame buffer should be congruent with that of the tiles with any other bank ID. Existence of such assignments is guaranteed because one such assignment can be achieved by placing the four tiles $4i, 4i+1, 4i+2$, and $4i+3$ in the same order in the double size tile labeled i for all the blocks. So far, the number of combinations is reduced to $(4!)^{k-1}$ since we have only to consider one quadruplet of banks (e.g., bank 0, 1, 2, and 3 if we choose $i = 0$) and apply the result to the other quadruplets.

Third, we reduce the number of blocks under consideration by assuming periodicity of assignments. Specifically, for an even n we search assignments that repeat every two blocks both horizontally and vertically. For an odd n, we search assignments that repeat every two blocks horizontally and four blocks vertically so that the both periods have the same length in tiles. In this way, the number of bank ID assignment combinations becomes $(4!)^3$ or $(4!)^7$ regardless of the size of the frame buffer. Assuming periodicity also yields some other advantages: the bank ID calculation is simplified as upper bits of a pixel coordinate become irrelevant to the bank ID; the equity property check and the uniformity evaluation can be performed for a constant number of tiles.

Thanks to the three constraints above, the number of possible bank ID assignments is at most $(4!)^7 \approx 4G$, which is tractable and independent of N. And the total time complexity is $O(n)$ due to recursion. Although the assignments obtained in this way are not guaranteed to be optimal, it is guaranteed that uniformity of the assignment for a larger N is equal to or greater than that for a smaller N.

Figure 6 shows the assignments obtained up to $N = 32$. The bank ID for N, denoted by $bankID_N$, is calculated from tile coordinates (t_x, t_y) by the following equations, where tile coordinates are pixel coordinates divided by the tile size.

$$
\begin{aligned}
bankID_1 &= 0 & bankID_2 &= t_x[0] \oplus t_y[0] \\
bankID_4[1] &= t_y[0] & bankID_4[0] &= t_x[0] \oplus t_y[1] \\
bankID_8[2] &= t_x[1] \oplus t_y[1] \\
bankID_8[1] &= ((t_y[1] \wedge (\overline{t_x[1]} \oplus t_x[0])) \vee (\overline{t_y[1]} \wedge t_y[0])) \oplus t_x[2] \oplus t_y[2] \\
bankID_8[0] &= ((t_y[1] \wedge (\overline{t_x[1]} \oplus t_y[0])) \vee (\overline{t_y[1]} \wedge t_x[0])) \oplus t_x[2] \oplus t_y[2] \\
bankID_{16}[3] &= t_y[1] & bankID_{16}[2] &= t_x[1] \oplus t_y[2] \\
bankID_{16}[1] &= t_y[0] \oplus t_x[2] \oplus (t_y[2] \wedge (\overline{t_x[0]} \oplus t_x[1])) \\
bankID_{16}[0] &= t_x[0] \oplus t_y[2] \\
bankID_{32}[4] &= t_x[2] \oplus t_y[2] \\
bankID_{32}[3] &= ((t_y[2] \wedge (\overline{t_x[2]} \oplus t_x[1])) \vee (\overline{t_y[2]} \wedge t_y[1])) \oplus t_x[3] \oplus t_y[3] \\
bankID_{32}[2] &= ((t_y[2] \wedge (\overline{t_x[2]} \oplus t_y[1])) \vee (\overline{t_y[2]} \wedge t_x[1])) \oplus t_x[3] \oplus t_y[3] \\
bankID_{32}[1] &= t_y[0] & bankID_{32}[0] &= t_x[0]
\end{aligned}
$$

where "$\overline{\cdot}$", "$\wedge$", "$\vee$", and "$\oplus$" mean logical NOT, AND, OR, and XOR operations, respectively, and expression $a[i]$ denotes the ith least significant bit of a. These equations can be derived by logic reduction.

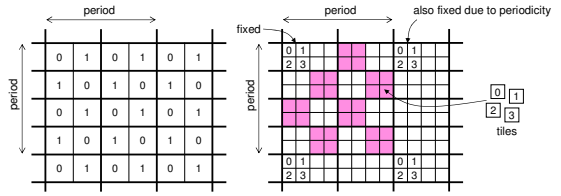


Figure 5: *Reduced search space for $N = 8$. (a) The assignment for $N = 2$ in double size. (b) In each of the tiles labeled 0 in (a), we place four tiles 0-3. In each of the tiles labeled 1 in (a), we place four tiles 4-7. We have only to consider either of these two quadruplets of tiles (we choose 0-3). We fix the assignment in one block (in this figure the upper left block), and consider placing tiles in the painted area in seven blocks. The assignment for the other blocks is determined automatically thanks to the periodicity.*

4. Evaluation Method

This section describes a method we adopt in order to evaluate performance of various storage schemes including ours.

4.1. Evaluation Model

Evaluating storage schemes based on numerous types of graphics system architectures and their possible combinations of parameters is impractical. Therefore, we use a

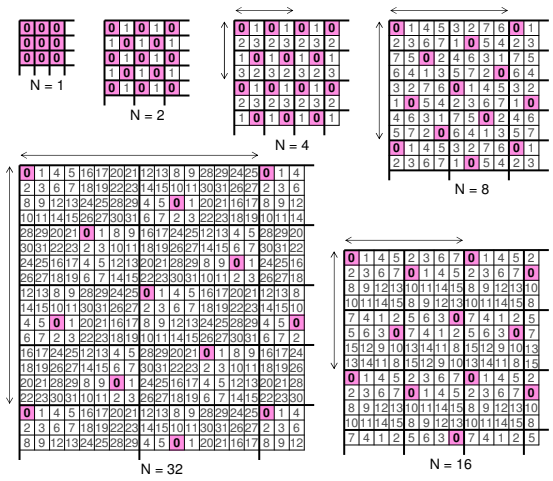


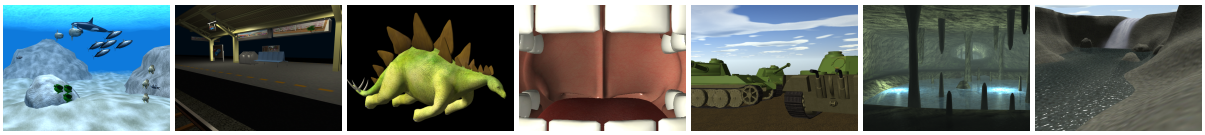
Figure 6: *Bank ID assignments for up to $N = 32$. Hexagonal patterns appear for $N \geq 4$, though when $N = 16$ the pattern also appears pentagonal or heptagonal.*

generic model shown in Figure 2(d). In this model, a stream of tiles is input, and each tile is sent, via the sorting network, to the appropriate memory bank depending on its tile coordinate. Each bank has a tile FIFO that can smooth out some of the imbalance in the number of input tiles.

For frame buffers, a tile stream is generated by the rasterizer. From the viewpoint of the memory banks, it is only the order of input tiles that affects load balancing. Therefore we set aside rasterization parallelism and types of architecture, and change the order of input tiles only by algorithms used by a single rasterizer. For this purpose, we chose the following three rasterization algorithms as representatives. The rasterization granularity is tiles rather than pixels.

- **RowMajorOrder**: This classical algorithm rasterizes from the top of a triangle toward right along a scanline, and returns to the left edge on the next scanline.
- **BlockedOrder**: This algorithm traverses blocks in row-major order and rasterizes within each block again in row-major order. This improves frame buffer and texture access locality [MM00].
- **HilbertOrder**: This algorithm rasterizes along a Hilbert curve [MWM01], which has high spatial locality.

We evaluate performance of storage schemes with and without a frame buffer cache. When the cache is enabled, the tile stream generator outputs a tile only when a cache miss occurs. The evaluation without a cache serves as an analysis of load imbalance of multiple graphics processors in sort-first and sort-middle architectures because a pair of a graphics processor and a memory bank (R, F, and B) in Figure 2(a) can be collectively regarded as a memory bank (B) in Figure 2(d), and a cache does not exert a strong influence over the graphics processors' load when a heavy pixel shader is used.



Sea Station Stegosaurus Mouth Tank Cave Waterfall

Figure 8: *Scenes used to evaluate performance of storage schemes.*

For textures, on the other hand, a tile stream is generated by the texture unit. Similar to the case of frame buffers, we do not deal with texturing parallelism. A single rasterizer generates a stream of tiles of fragments, and each fragment provides a texture coordinate to the texture unit. The texture unit has a cache, and when a cache miss occurs, it fetches the tile containing the specified texel from the appropriate memory bank (in this sense we had better use the term "tile *request* stream"). Again, variations of the order of tiles in a tile request stream is controlled by rasterization algorithms.

Based on the previous work [HG97, IEP98, IEH99], we use a 16KB two-way set associative cache with an LRU replacement policy and with a cache line size equal to the data size of a tile. We calculate the address A of a tile as follows.

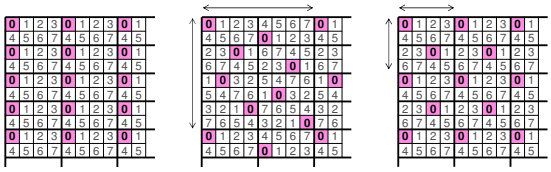
$$A = B + (blockID \cdot N + bankID) \cdot 4T^2$$

where B is the base address, and T is the tile size. We set the data size of a pixel/texel to be 4 bytes.

4.2. Counterpart Storage Schemes

In order to evaluate how much our hexagonal storage scheme improves memory access load balance, we compare it with the three storage schemes described below. An example of the assignments for each scheme is shown in Figure 7.

- **Rectangular**: In this commonly used storage scheme, the assignment does not change block to block, and the pattern is rectangular.
- **MFB**: The aim of the Multiaccess Frame Buffer [Har94] is to assign N banks so that no bank ID coincides in any rectangle whose area is $N/2$. Though some pairs of tiles having the same ID are close to each other contrary to this aim [Wei96], the assignment is fairly uniform.
- **Flipped**: This storage scheme is based on Rectangular scheme, but it improves uniformity by simply flipping bank IDs in the left half of a block and those in the right half for every other row of blocks.



Rectangular MFB Flipped

Figure 7: *Counterpart bank ID assignments for* $N = 8$.

4.3. Scenes

We used the seven scenes shown in Figure 8. All primitives are texture mapped with mipmaps. Thus each fragment requires eight texels under texture minification, and four texels under magnification. The frame buffer size is 512×512.

5. Results

In order to limit the number of combinations of parameters, we show the results for N = 8, 16, and 32 with a tile size of 4×4 pixels. For $N < 8$, the assignments of our storage scheme are equivalent to those of Rectangular or Flipped scheme. A frame buffer cache is enabled for the following results if not otherwise specified. From experiment, both of the frame buffer and texture cache hit ratios are almost independent of storage schemes.

Figure 9 shows the overall imbalance of the number of accesses to the memory banks for frame buffers and textures. Thanks to fine interleaving, frame buffer accesses are evenly distributed to the banks regardless of storage schemes, although some variations can be seen for Stegosaurus scene. On the other hand, texture access load imbalance is relatively large, and is dependent on storage schemes for most of the scenes. This is because textures are accessed more irregularly. We can see that the overall imbalance of texture accesses for Hexagonal scheme is almost always smaller than that for Flipped and Rectangular schemes, and it is competitive (smaller for some scenes, and larger for some scenes) with that for MFB scheme.

Even if the overall imbalance is small, there are possibilities of strong temporal imbalance. Thus we measure imbalance of the number of accesses to the memory banks during some period of time, assuming that the tile stream generator provides one tile per cycle. Figure 10 shows the time development of such imbalance for Mouth scene as an example. The temporal imbalance of the frame buffer accesses can exceed 150% for all the storage schemes contrary to the overall balanced memory accesses. For both frame buffers and textures, we can see that the graph for Rectangular scheme has higher peaks than those for the other storage schemes.

Since we cannot trace the whole time development of the temporal imbalance for each scene, we count the intervals in cycles between two consecutive tiles that a memory bank receives. Figure 11 shows the histograms of such intervals. The histograms for the frame buffer have some spikes that

correspond to the spatial intervals between the tiles having the same bank ID along the trace of a given rasterizer. The histograms for the textures are smoothed out since textures are traversed more arbitrarily. For both frame buffers and textures, the histogram for Rectangular scheme is strongly shifted to the left, and those for MFB and Flipped schemes are slightly shifted to the left compared to that for Hexagonal scheme. High occurrence of small intervals suggests there would be high peak memory access load imbalance. We take the standard deviation of the intervals as a measure of the temporal imbalance for the entire scene, which is shown later.

We also estimate the performance degradation due to bank conflicts. We assume that each of the N memory banks is busy for N cycles after receiving a tile. The system is balanced because the memory has potential ability to receive N tiles in N cycles while the tile stream generator outputs N tiles. If another tile is sent to a busy memory bank, it is queued in the tile FIFO of that bank. If the FIFO is full, the tile stream generator stalls. The performance measure is the total cycles required to render the entire scene divided by the number of tiles. Figure 12 shows the performance degradation for Mouth scene. Naturally, the values decrease as the number of FIFO stages increases, but the ratio of the value for one storage scheme to that for another is approximately maintained. The graphs show that Hexagonal scheme has the least performance degradation of the four schemes.

Figure 13 summarizes the standard deviation of the intervals and the performance degradation for all combinations of the storage schemes, scenes, rasterization algorithms, and the number of memory banks. The standard deviation is normalized by N for comparison. A graph of the standard deviation and the corresponding graph of the performance degradation are similar in shape, which supports the use of the standard deviation of tile intervals as a measure of temporal load imbalance. As the overall tendency, the lines in the graphs slope upward from left to right, meaning that Hexagonal scheme is less subject to temporal load imbalance than the other storage schemes are. The major exception to this tendency is the relationship between Hexagonal and MFB schemes when $N = 16$, which can be attributed to the fact that the assignment of Hexagonal scheme for $N = 16$ is not quite close to regular hexagons as shown in Figure 6.

Finally, Table 1 summarizes the performance gain obtained by using Hexagonal scheme over the other storage schemes. We evaluate the number of cycles reduced by using Hexagonal scheme relative to the total cycles required by using a given storage scheme, and simply average these values for all the scenes. These figures show that Hexagonal scheme can improve performance by an average of 10% compared to Rectangular scheme, and that it is advantageous over MFB and Flipped schemes except for MFB for $N = 16$, though the differences may be small in some cases.

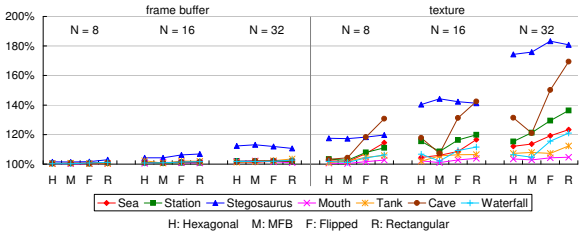


Figure 9: *Overall imbalance of memory accesses. The y-axis shows the percentage of the maximum number of tiles given to a memory bank to the average number of tiles per bank. This graph is for a RowMajorOrder rasterizer, but the graphs for the other two rasterizers have a similar tendency.*

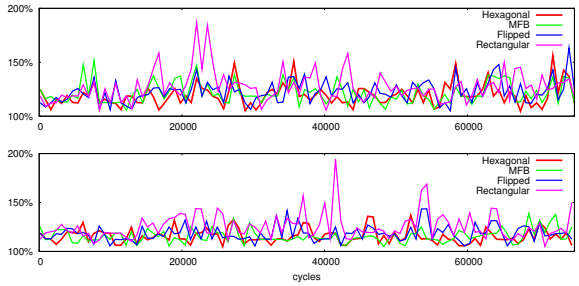


Figure 10: *Time development of temporal imbalance of memory accesses to the frame buffer (top) and to the textures (bottom) for Mouth scene, $N = 8$, and HilbertOrder. Each data point is windowed by 128 cycles.*

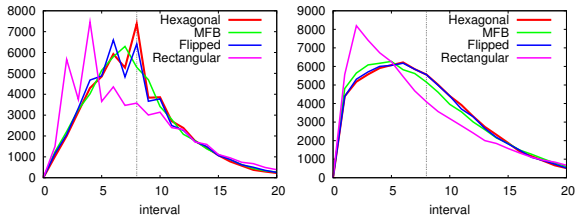


Figure 11: *Histograms of the intervals between two consecutive tiles sent to the same memory bank of the frame buffer (left) and of the textures (right) for Mouth scene, $N = 8$, and HilbertOrder. The vertical lines show the ideal interval of 8.*

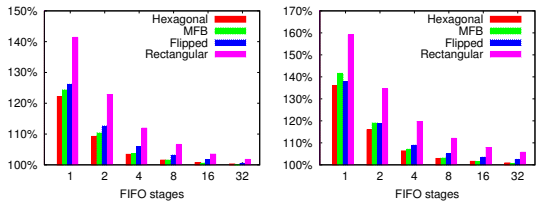


Figure 12: *Performance degradation due to imbalance of memory accesses to the frame buffer (left) and to the textures (right) for Mouth scene, $N = 8$, and HilbertOrder. Several variations of the number of FIFO stages are shown. The y-axis shows the percentage of the number of cycles required to render the entire scene to the total number of tiles.*

Table 1: *Performance gain obtained by using the hexagonal storage scheme over the other three storage schemes.*

	Compared to	$N = 8$	$N = 16$	$N = 32$
Frame buffer (with cache)	MFB	1.1%	−8.6%	0.2%
	Flipped	6.3%	10.4%	8.2%
	Rectangular	11.5%	11.7%	14.4%
Frame buffer (w/o cache)	MFB	1.9%	−3.6%	2.8%
	Flipped	3.8%	7.4%	4.5%
	Rectangular	10.3%	9.3%	10.6%
Texture	MFB	3.0%	−0.3%	0.6%
	Flipped	3.1%	3.3%	1.6%
	Rectangular	11.2%	7.2%	11.9%

6. Conclusion

This paper has presented a storage scheme which statically assigns pixel/texel coordinates to multiple memory banks, conforming to the shape of nearly regular hexagons. Using several typical 3D graphics scenes, we have shown that our storage scheme distributes memory access load over the memory banks more evenly than the other representative schemes. Although the performance gain is modest (even marginal compared to MFB scheme), the results support the validity of our strategy for reducing bank conflicts.

Our hexagonal bank ID assignment appears complex, but thanks to the power-of-two block size and repeat period, bank ID calculation can be simplified. It requirs only a few additional logical operations, if any, compared to the other storage schemes, which has a minimal impact upon silicon area in hardware implementation.

In future work, we intend to ascertain whether better assignments exist for large N. Also, we intend to consider some other possible applications of our hexagonal assignment, such as antialiasing.

Acknowledgments

We would like to thank Prof. Tomoyuki Nishita and Shigeo Takahashi (The University of Tokyo) for their useful suggestions. We would also like to thank Shingo Yanagawa, Katsundo Nagashima, and Satoyuki Inaba for the scenes.

References

[AMH02] AKENINE-MÖLLER T., HAINES E.: *Real-Time Rendering (second edition)*. A K Peters, Ltd., 2002. 3

[CS98] CONWAY J. H., SLOANE N. J. A.: *Sphere Packings, Lattices and Groups (3rd edition)*. Springer-Verlag, New York, 1998. 3

[Dee93] DEERING M. F.: Data complexity for virtual reality: where do all the triangles go? In *IEEE Virtual Reality Annual International Symposium* (1993), pp. 357–363. 2

[Fuc77] FUCHS H.: Distributing a visible surface algorithm over multiple processors. In *the ACM Annual Conference* (1977), pp. 449–451. 2

[FvDFH90] FOLEY J. D., VAN DAM A., FEINER S. K., HUGHES J. F.: *Computer Graphics: Principles and Practice (2nd edition)*. Addison-Wesley, 1990. 2

[Gla94] GLASSNER A. S.: *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, Inc., 1994. 2

[Har94] HARPER, III D. T.: A multiaccess frame buffer architecture. *IEEE Trans. Comput. 43*, 5 (May 1994), 618–622. 2, 5

[HG97] HAKURA Z. S., GUPTA A.: The design and analysis of a cache architecture for texture mapping. In *the 24th Annual International Symposium on Computer Architecture* (1997), pp. 108–120. 5

[IEH99] IGEHY H., ELDRIDGE M., HANRAHAN P.: Parallel texture caching. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (1999), pp. 95–106. 2, 5

[IEP98] IGEHY H., ELDRIDGE M., PROUDFOOT K.: Prefetching in a texture cache architecture. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (1998), pp. 133–142. 5

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE CG&A 14*, 4 (July 1994), 23–32. 2

[MM00] MCCORMACK J., MCNAMARA R.: Tiled polygon traversal using half-plane edge functions. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2000), pp. 15–21. 4

[Mue95] MUELLER C.: The sort-first rendering architecture for high-performance graphics. In *the Symposium on Interactive 3D Graphics* (1995), pp. 75–85. 2

[MWM01] MCCOOL M. D., WALES C., MOULE K.: Incremental and hierarchical Hilbert order edge equation polygon rasterization. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2001), pp. 65–72. 4

[NK96] NISHIMURA S., KUNII T. L.: VC-1: A scalable graphics computer with virtual local frame buffer. In *SIGGRAPH 96* (1996), pp. 365–372. 2

[Par80] PARKE F. I.: Simulation and expected performance analysis of multiple processor z-buffer systems. *Computer Graphics 14*, 3 (July 1980), 48–56. 2

[PH89] POTMESIL M., HOFFERT E. M.: The Pixel Machine: A parallel image computer. *Computer Graphics 23*, 3 (July 1989), 69–78. 2

[Tyt00] TYTKOWSKI K. T.: Hexagonal raster for computer graphic. In *IEEE International Conference on Information Visualization 2000* (2000), pp. 69–73. 2

[Wei96] WEI B.: Comments on "a multiaccess frame buffer architecture". *IEEE Trans. Comput. 45*, 7 (July 1996), 862. 2, 5

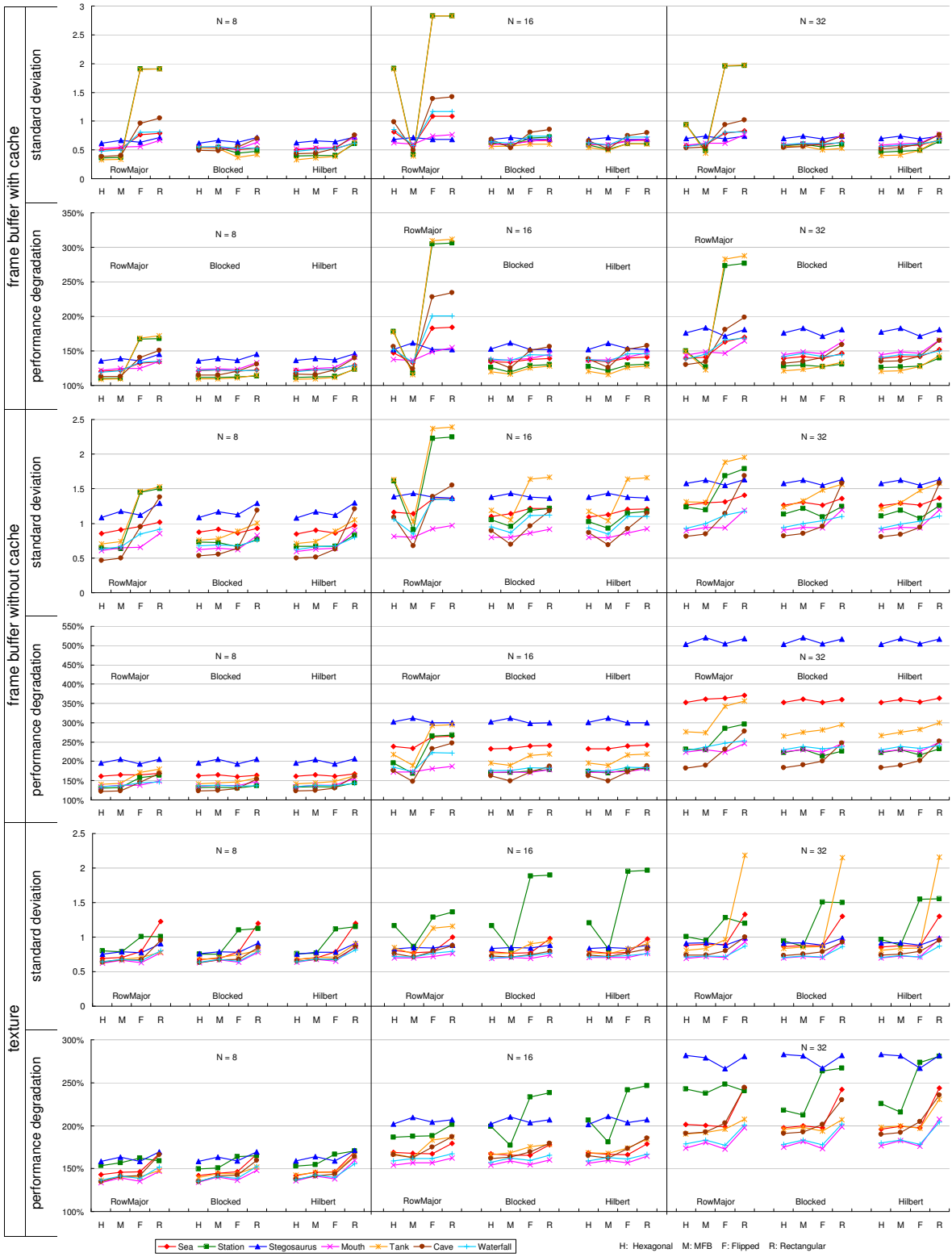


Figure 13: *Normalized standard deviation of the intervals between two consecutive tiles sent to the same memory bank, and the performance degradation due to imbalance of memory accesses with one FIFO stage.*