A *Mid-texturing* Pixel Rasterization Pipeline Architecture for 3D Rendering Processors

Woo-Chan Park, Kil-Whan Lee, Il-San Kim, Tack-Don Han, and Sung-Bong Yang
Media System Laboratory, Department of Computer Science
Yonsei University, Seoul, Korea
{chan, kiwh, sany, hantack}@kurene.yonsei.ac.kr, yang@mythos.yonsei.ac.kr

Abstract

As a 3D scene becomes increasingly complex and the screen resolution increases, the design of effective memory architecture is one of the most important issues for 3D rendering processors. We propose a pixel rasterization architecture, which performs a depth test operation twice, before and after texture mapping. The proposed architecture eliminates memory bandwidth waste caused by fetching unnecessary obscured texture data, by performing the depth test before texture mapping. The proposed architecture reduces the miss penalties of the pixel cache by using a pre-fetch scheme — that is, a frame memory access, due to a cache miss at the first depth test, is done simultaneously with texture mapping. The proposed pixel rasterization architecture achieves memory bandwidth effectiveness and reduces power consumption, producing high-performance gains.

1. Introduction

In current high-performance rendering processors, not only basic functions, such as scan-conversion and *z* (depth)-test, but also realization mappings, such as texture mapping, bump mapping, and environment mapping, are provided to enhance the realism and the visual complexity of computer generated images [2], [7], [8]. Texture mapping is one of the especially important criteria in estimating the performance of a rendering processor. To perform texture mapping at a high pixel fill rate, considerable hardware logic is required and heavy memory traffic is also generated [5], [9], [10], [11]. The heavy memory traffic results in the memory-bandwidth problem, which needs to be resolved for the pixel rasterization pipeline to run fully at a higher pixel fill rate. Therefore, organizing the texture mapping stage effectively is one of the major design issues with high-performance rendering processors [3], [4], [5], [9], [10], [11].

In conventional high-performance rendering processors, texture mapping is performed before the *z*-test [2], [4], [5], [6], [7], [8], [9], [10]. We call this processing flow *pre-texturing*. *Pre-texturing* supports well the semantics of standard APIs such as OpenGL [12], [13]. The major disadvantage of *pre-texturing* is unnecessary texture mapping for the fragments obscured by the previously drawn pixel. These unnecessary operations may cause serious memory bandwidth waste and thus result in considerable degradation in the overall performance.

In order to remove such unnecessary operations, texture mapping is performed after the *z*-test. We call this form of processing *post-texturing*. However, a wider fragment queue for the pipeline execution is required because both the current fragment information and the retrieved data from the frame memory with respect to the current pixel address should be processed with the fragment queue. The fragment queue is essential for keeping the execution of the

pipeline running fully at a higher pixel fill rate, and it plays an important role in a texture pre-fetching scheme [10]. Moreover, the wide separation between reading and writing a *z*-value cannot be avoided because the OpenGL semantics for the transparency of texture images should be supported. Such wide separation makes maintaining the frame memory consistency more difficult because more than one fragment may have the same pixel address.

In Radeon, the hierarchical *z* is one of the Hyper^{TM} technologies [2]. This keeps the reduced resolution of the *z*-buffer on the hierarchical *z*-buffer and removes the fragments with *z*-test failures as early as possible; they are removed from the pipeline before texture mapping. After this step, texture mapping and the final *z*-test with the full screen frame memory are performed. Consequently, two depth tests between the hierarchical *z*-buffer and the full screen frame memory are done for a fragment. With this scheme, a considerable number (60~70% on average) of fragments with *z*-test failures are detected and then discarded from the pipeline. However, the hierarchical *z* requires a large data structure. Maintaining the hierarchical *z* for every frame memory update may also produce an excessive computational burden.

In the proposed architecture, two new stages, *z-read* and *z-test*, are added to the pixel rasterization pipeline. Because texture mapping is performed between the first *z*-test and the second *z*-test, we call this processing *mid-texturing*. In the first *z*-test, the depth test is performed if the result of the pixel cache access for the first *z*-read is a hit. The second *z*-read and *z*-test are performed after texture mapping. In the proposed architecture, the unnecessary execution of texture mapping for obscured fragments is eliminated because texture mapping is performed after the first *z*-test as in *post-texturing*. However, unlike *post-texturing*, a normal sized fragment queue is sufficient and the consistency problem due to the wide separation between the first *z*-read and the *z*-write stages does not occur because the second *z*-read is performed after texture mapping. Additionally, in the case of a pixel cache miss at the first *z*-read stage, the frame memory access for the cache miss penalty can be performed simultaneously with the pipeline operations between the first and the second *z*-read stages. Therefore, the penalty of a pixel cache miss is drastically reduced.

We have built a trace-driven simulator for the proposed pixel rasterization architecture. To validate our proposed scheme, various simulation results with three benchmarks are given. The depth complexity and the *z*-test failure rate are also described. For various wide separation cases, the consistency problem occurrence rates are obtained. These rates are so low that unnecessary executions of texture mapping are negligible. Various simulations for the pixel cache hit rates with several cache organizations are also performed. Finally, the simulation results for the cache miss penalty reduction of the proposed architecture are presented.

In the next section, we give a brief overview of the conventional pixel rasterization pipeline flow and its architectural features. In Section 3, we illustrate a new rasterization architecture and its architectural features. Various simulation results are given in Section 4. Conclusions are presented in Section 5.

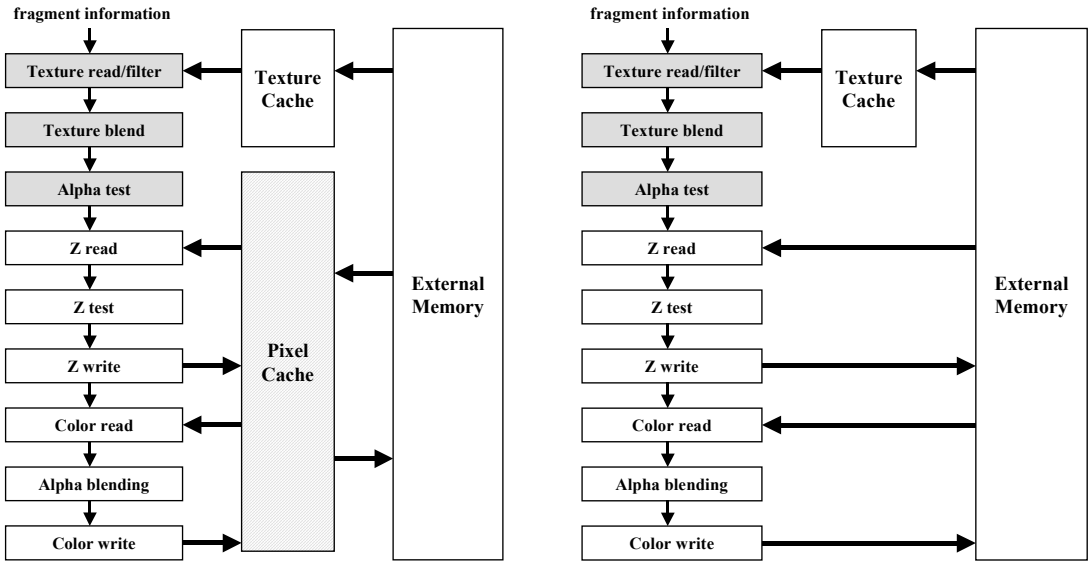
2. Background

The rasterization pipeline consists of three steps: per-triangle processing, per-span processing, and per-pixel processing. The per-triangle step, called triangle setup [8] for initialization, includes the computations of a series of increments used to walk along the edges of each triangle, and for the span interpolation. In the per-span step, called the edge-walk, the span endpoints and the interpolation parameters for pixel operations are

computed. In the per-pixel step, the pixel rasterization generates a series of fragments along the span by interpolating the color and texture coordinates.

2.1. The *Pre-texturing* Pixel Rasterization Architecture

Two types of detailed *pre-texturing* pixel rasterization pipelines are shown in the above figure; Fig. 1(a) includes the pixel cache and Fig. 1(b) does not. Most current rendering systems have a structure similar to one of these two types. The first two stages read four or eight texels from the texture cache, perform either bi-linear filtering or tri-linear filtering with them to produce a single texel, and blend the texel with the pixel color. The *alpha*-value of the current fragment is then compared with that of the filtered texel. The *alpha*-test decides whether a fragment is accepted or not, based on the comparison between the *alpha*-value of the incoming fragment and a reference value. If the test rejects a fragment, which means the *alpha*-test *fails*, (when either a fragment or a texel is completely transparent), then the fragment is dropped from the pipeline. The next two stages read the *z*-value from the frame memory or from the pixel cache and compare it with that of the current fragment. If the result of the *z*-test is a failure (that is, the current fragment is obscured by the previously drawn pixel), then it is removed from the pipeline. Otherwise, a new *z*-value is written into the frame memory or into the pixel cache. Finally, we read the color data, *alpha*-blend them with the result of texture blending, and then write the final color data back to either the frame memory or the pixel cache.



(a) with the pixel cache **(b) without the pixel cache**

Figure 1. The *pre-texturing* pixel rasterization pipeline

We define *depth complexity* as the average number of generated fragments per pixel for a frame — that is, depth complexity indicates how many objects are overlapped in the same pixel on average. For some scenes, depth complexity may be as high as 10 or more. In most cases, depth complexity is two or three. In a scene with a depth complexity of three, about seven out of eighteen fragments fail the *z*-test [5].

The documentation on *Neon* [5], [6] is very important for our research because no other documents on the internal architecture of a recent high-performance rendering processor have been made public. In [5], it is mentioned that *pre-texturing* is adopted for the following three reasons: First, *post-texturing* requires a wider fragment queue

for texture mapping after the *z*-test, and *Neon* could not afford it. Second, OpenGL semantics do not allow updating the *z*-buffer until texture mapping is finished, because a textured fragment may be completely transparent. Such a wide separation between reading and writing a *z*-value may cause difficulty in maintaining the frame memory consistency. Third, maintaining the spatial locality of texture access is difficult. However, since current semiconductor technology can afford sufficient cache sizes, this problem may not significantly affect the performance of a texture-cache system.

2.2. The *Post-texturing* Pixel Rasterization Architecture

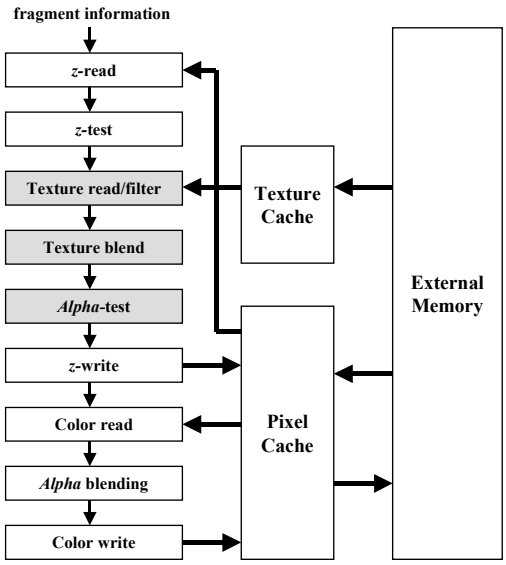


Figure 2. The *post-texturing* pixel rasterization architecture

The *post-texturing* pixel rasterization pipeline with pixel cache is shown in Fig. 2. The first stage reads the *z*-value from the pixel cache. When a pixel cache miss occurs for the *z*-read, the pipeline stops until the cache block of the cache miss is transmitted from the frame memory to the pixel cache. Next, the *z*-test is performed by comparing the *z*-value retrieved at the *z*-read stage with that of the current fragment. If the test fails, the current fragment is dropped from the pipeline. The next three stages, texture read/filter, texture blend, and *alpha*-test stages, are identical to those in the *pre-texturing* architecture. Finally, we read the color data, *alpha*-blend them, and then write them back to the pixel cache.

Since the *z*-data retrieved from the pixel cache at the *z*-read stage should be processed up to the *z*-write stage, along with the fragment information, a wider fragment queue is needed between the *z*-read and the *z*-write stages. Since recent rendering processors are able to generate four pixels in a cycle, if the pipeline length between the *z*-read and the *z*-write stages is at least twenty, then a queue size of at least 3K bits should be added to the normal fragment queue. However, this hardware burden is not negligible.

The wide separation between reading and writing *z*-values in *post-texturing* causes the consistency problem. If two fragments have the same pixel address, then the *z*-write for the first fragment must be completed before the *z*-read of the second fragment. Since such a close time overlap rarely occurs, it is acceptable to stop reading the pixel data until the *z*-write for the first fragment is complete. However, an associative overlap detector for this separation is not suitable because of its hardware burden.

3. The Proposed Pixel Rasterization Architecture

Fig. 3 shows the proposed pixel rasterization pipeline with *mid-texturing*. Compared with either *pre-texturing* or *post-texturing*, extra *z*-read and *z*-test stages are added to the pipeline. The first pair of *z*-read and *z*-test operations is performed before texture mapping, and the second pair is performed after texture mapping. The proposed architecture has the following two advantages. First, the architectural advantages of both *pre-texturing* and *post-texturing* can be obtained. Since texture mapping is performed after the first *z*-test, there is no memory bandwidth waste caused by fetching obscured texture data. Since the second pair of *z*-read and *z*-test operations is performed after texture mapping, a normal-sized fragment queue is sufficient. Even though the consistency problem may occur between the first *z*-read and the *z*-write stages, the consistency between the frame buffer and pixel cache can be maintained because the second *z*-read and *z*-test are performed after texture mapping. Second, the penalty of a pixel cache miss is drastically reduced. The additional hardware for the first *z*-read stage, such as one read port for the *z*-data of the pixel cache, is required for the pixel cache. This hardware overhead is tolerable thanks to advances in current semiconductor technology. In addition, the memory bandwidth waste caused by an additional *z*-read is not significant because *z*-data are read from the pixel cache.

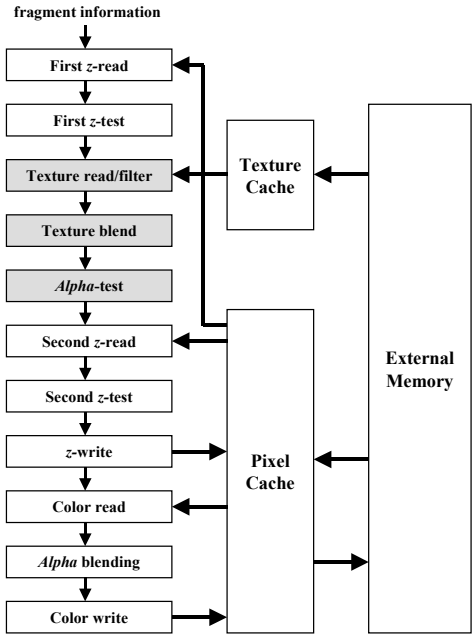


Figure 3. The proposed pixel rasterization pipeline: *mid-texturing*

We now describe the processing flow of the pipeline in Fig. 3. In the first stage the *z*-value is read. The pipeline does not stop its execution even if a pixel cache miss occurs, as opposed to *pre-texturing* and *post-texturing*. For a fragment with a cache miss, the depth test is performed by the second *z*-read and *z*-test operations after texture mapping. In case of a pixel cache miss, the miss penalty that transfers a missed block from the frame memory into the pixel cache can be handled simultaneously with the pipeline executions between the first *z*-read and the second *z*-read stages. With this scheme, we can reduce the miss penalty drastically. Our simulation results for the miss penalty reduction are provided in the next section.

After the first *z*-read stage, a *z*-test is performed in the case of a pixel cache hit. If the test fails then it is dropped from the pipeline. Therefore, the pipeline execution will proceed to the

next stage in the case of either a pixel cache miss or a *z*-test success for a pixel cache hit. Then texture mapping/filtering, texture blending, and the *alpha*-test are performed in turn. The second *z*-read and *z*-test are done afterwards. Next, a new *z*-value is written to the pixel cache according to the result of the *z*-test. The extra stages, such as color read, *alpha* blending, and color write are identical to the previous architectures.

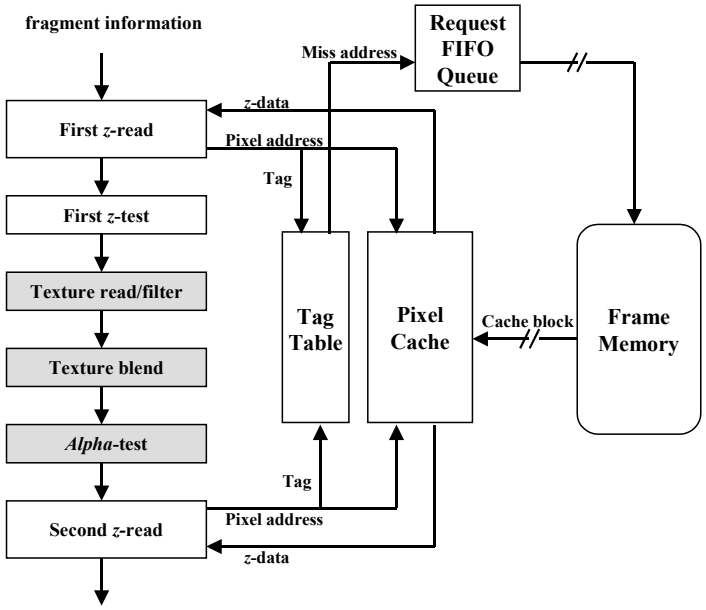


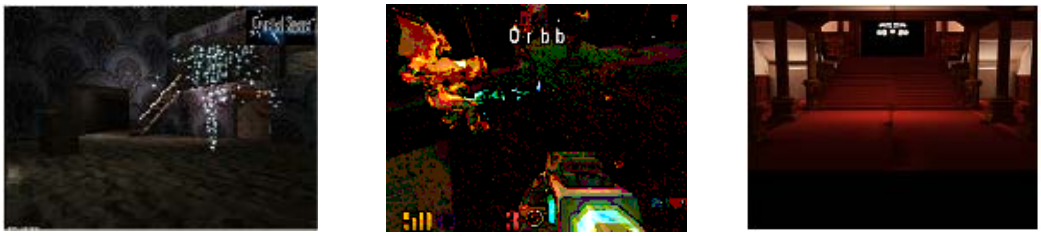
Figure 4. The pixel cache architecture in the pixel rasterization pipeline

Fig. 4 shows the detailed pixel cache architecture in the pixel rasterization pipeline. In the first *z*-read, the tag field of the pixel address of an input fragment is looked for in the tag table of the pixel cache. Then the pixel address of an input fragment and that of a cache tag are compared. If a tag comparison reveals a miss, the cache tag is updated with the pixel address of the fragment and then this address is forwarded to the memory request FIFO queue. The request FIFO queue sends a request for the missing cache blocks to the frame memory. Then the requested cache blocks are transmitted to the pixel cache. When a fragment reaches the second *z*-read, the tag is checked again. If its result is a hit, the pipeline execution can proceed immediately, otherwise it must wait until the corresponding cache block is completely transmitted from the frame memory. It may happen that the older cache block is prematurely overwritten with a new cache block. When the result of tag checking of the first *z*-read is a hit and that of the second *z*-read is a miss for a fragment, the full execution cycle for a cache miss penalty should be executed. However, the occurrence rate of this case is so small that it does not affect the performance of the pixel cache significantly. Our simulation results for the miss penalty reduction of the pixel cache are provided in the next section.

4. Experimental Simulations

In order to validate the proposed architecture, various simulation results are given in this section. A trace-driven simulator has been built for the proposed architecture. The traces are generated with three benchmarks, *Crystal Space*, *Quake3*, and *Lightscape*, by modifying the Mesa OpenGL compatible API (Application Programming Interface). For each benchmark, 100 frames are used to generate each trace. Fig. 5 shows the captured scenes for the benchmarks. *Crystal Space* and *Quake3* are OpenGL-based video game engines. *Quake3* in particular is typical of current video games and is frequently used as

a benchmark in other related works for their simulations. *Lightscape* is a product of SPECviewperf™, and is an industrial standard benchmark for measuring the 3D rendering performance of systems running under OpenGL. *Lightscape* is used as a benchmark in this paper because of its high scene complexity compared with other SPECviewperf™ products.

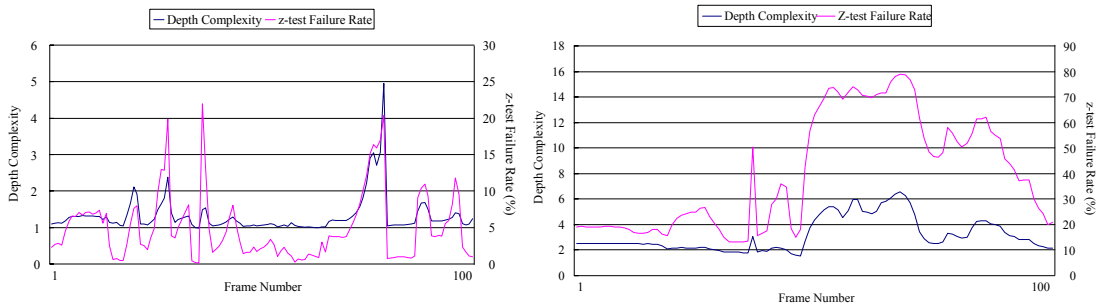


(a) Crystal Space **(b) Quake3** **(c) Lightscape**

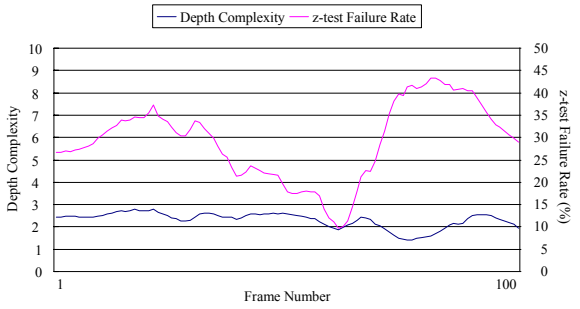
Figure 5. Three benchmarks

4.1. Depth Complexity versus *z*-test Failure Rate

Fig. 6 shows the depth complexity and the *z*-test failure rate for each benchmark as the frames are running. The depth complexity and the average *z*-test failure rate for each benchmark are given in Table 1. In most cases, the *z*-test failure rate increases as the corresponding depth complexity increases. If the scene complexity increases, the *z*-test failure rate also increases. The *z*-test failure rate also represents the quantity of redundant texture mapping of *pre-texturing*. Therefore, the memory bandwidth saving rate for texture mapping of *mid-texturing* equals the *z*-test failure rate minus the redundant executions rate.



(a) Crystal Space **(b) Quake3**



(c) Lightscape

Figure 6. Depth complexity versus *z*-test failure rate

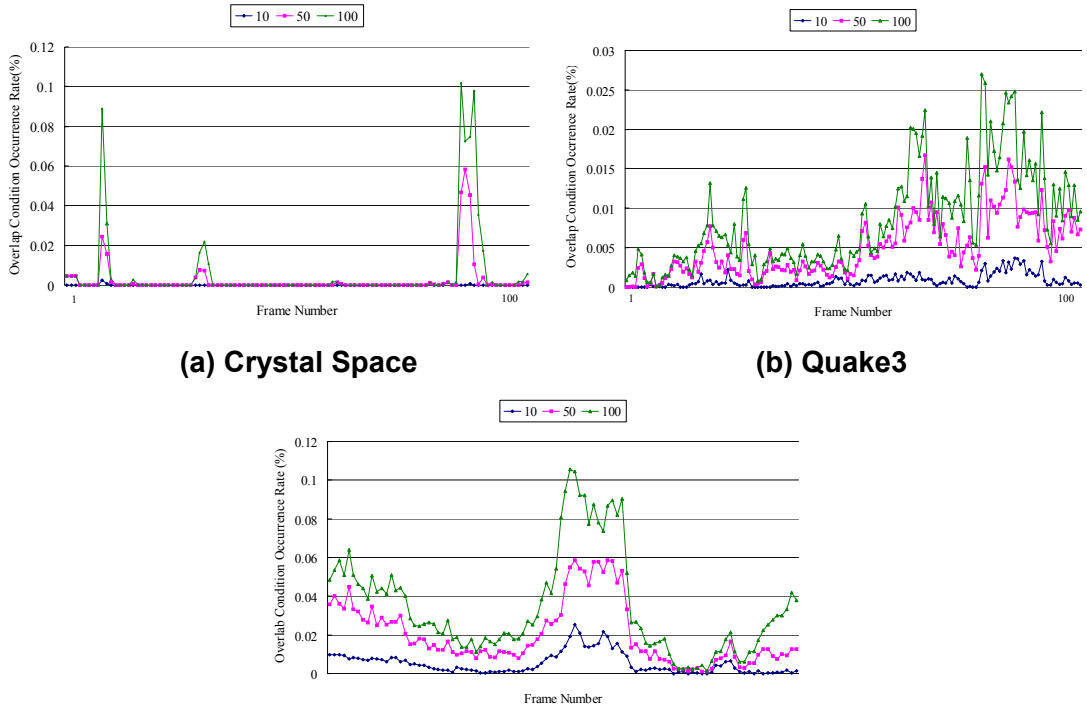
Three cases cause redundant executions. The first case is caused by the obscured fragments among those with pixel cache misses at the first *z*-read stage. The rate of this first case can be calculated by multiplying the pixel cache miss rate by the *z*-test failure rate, which is small

and hence negligible. The second is generated when the first *z*-test succeeds owing to some overlap occurrence, while the first *z*-test fails if the consistency is maintained between the first *z*-read and the *z*-write stages. Because the overlap occurrence rate (as described in Section 4.2) is so low, it can be neglected. The third is caused by an additional *z*-read operation. However, memory bandwidth wasted by an additional *z*-read is not significant because *z*-data are read from the pixel cache.

Table 1. Depth complexity and *z*-test failure rate

	Depth complexity	Average *z*-test failure rate
Crystal Space	1.31	4.93
Quack 3	3.18	39.76
Lightscape	2.13	29.16

4.2. Wide Separation versus Overlap Condition Occurrences



(a) Crystal Space **(b) Quake3**

(c) Lightscape

Figure 7. The overlap condition occurrence rate

Because of the wide separation between the *z*-read and the *z*-write stages, the consistency problem may occur owing to the overlap condition. Fig. 7 shows the rate of occurrence of the overlap condition when the separations are 10, 50 and 100 wide. It assumes that the geometry processing unit, the frame memory system, and the texture mapping memory system are perfect — that is, the latency of each unit is assumed to be zero. Thus it can be assumed that one pixel is generated per cycle with one pixel pipeline. The simulation results show that the performance degradation due to the wide separation is quite low because the overlap conditions occur very rarely.

4.3. Pixel Cache Hit Ratio and Cache Miss Penalty Reduction

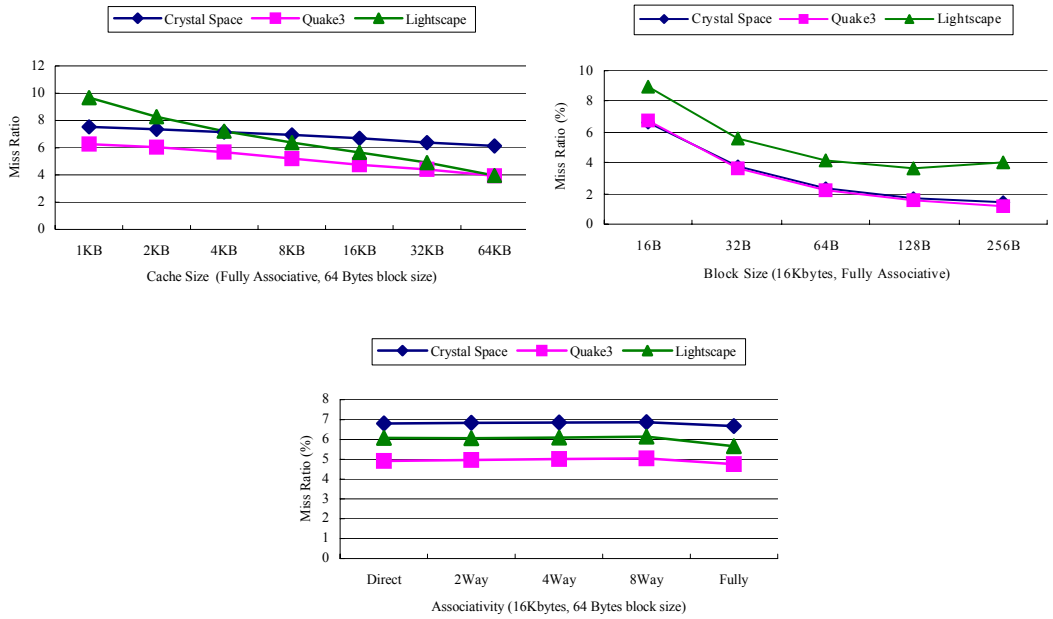


Figure 8. The pixel cache miss rates according to cache size, block size, and associativity

Fig. 8 shows the cache miss rates with a conventional pixel cache architecture having various cache sizes, block sizes, and set associativities. The simulation results show that the miss rate varies according to the block size, not to the cache size and the associativity. As the block size increases, the miss rate decreases.

If the cache miss rate is 5% and the miss penalty takes 10 cycles, the performance is degraded by about 35%. As semiconductor technology advances, this performance degradation also increases due to the increase in miss-penalty cycles. Moreover, the performance degradation caused by the cache miss penalty is also unavoidable.

Fig. 9 shows the cache miss penalty reductions by *mid-texturing* with various wide separations for a direct mapped 16Kbyte cache with a 64-byte block size. We assume that the miss penalty can be handled in 10 cycles, considering the clock frequencies of a rendering processor and a high-performance memory, such as a Double Data Rate memory. Even though the pixel cache miss rates are similar for each benchmark, as shown in Fig. 8, the miss penalty reductions vary because of their cache miss distributions—for example, several groups of misses occur in *Lightscape*. This assumes that the geometry processing unit and the texture mapping memory system are perfect. A perfect zero-latency frame memory system can be achieved if the pixel cache is hit perfectly or if the miss penalty is reduced to zero. Fig. 9 shows that the miss penalty reduction rates decrease more or less when the length of the wide separation is over 50 due to premature overwriting the older cache block with a new one as described in the previous section. However, the decrease in the miss penalty reduction may not be significant because the stage separation of 50 is sufficient for real processor implementation.

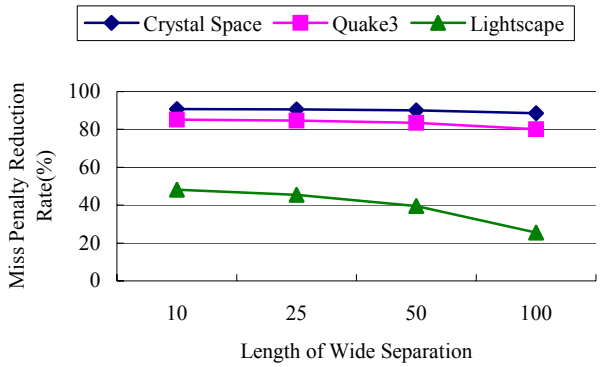


Figure 9. Cache miss penalty reductions by mid-texturing

5. Conclusion

Our proposed pixel rasterization pipeline can provide memory bandwidth effectiveness and performance improvement. An accurate dynamic cycle simulator for the proposed architecture is now under development. An effective realization unit (including bump mapping, environment mapping, and image warping) is also being studied. The proposed rendering processor will be implemented as a chip in near future.

References

[1] J.D. Foley, A. Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics*, *Principles and Practice*, Second Edition, Addison-Wesley, Massachusetts, 1990.

[2] S. Morein, "ATI Radeon - Hyper*Z* technology," *Hot3D Session of 2000 Eurographics Workshop on Computer Graphics Hardware*, http://www.merl.com/hwws00/presentations/ATIHot3D.pdf, Aug. 2000.

[3] D. Kirk, "Unsolved problems and opportunities for high-quality, high-performance 3-D graphics on a PC platform," *Proceedings of SIGGRAPH/Eurographics Workshop on graphics hardware*, Keynote talk, Aug. 1998.

[4] J.S. Montrym, D.R. Baum, D.L. Dignam, and C.J. Migdal, "InfinityEngine graphics," *Proceedings of SIGGRAPH '97*, pp. 293-302, Aug. 1997.

[5] J. McCormack, R. McNamara, C. Gianos, L. Seiler, N. P. Jouppi, K. Correl, T. Dutton, and J. Zurawski, "Neon: a (big) (fast) single-chip 3D workstation graphics accelerator," Research Report 98/1, Western Research Laboratory, Compaq Corporation, Aug. 1998 (revised July 1999).

[6] J. McCormack, R. McNamara, C. Cianos, N. P. Jouppi, T, Dutton, J. Zurawski, L. Seiler, and K. Corell, "Implementing Neon: A 256-bit graphics accelerator," *IEEE Micro*, vol. 19, no. 2, April 1999, pp. 58-69.

[7] T. Ikedo and J. Ma, "The Truga001: A scalable rendering processor," *IEEE computer and graphics and applications*, vol. 18, no. 2, March 1998, pp. 59-79.

[8] A. Kugler, "The setup for triangle rasterization," *Proceedings of 11th Eurographics Workshop on Computer Graphics Hardware*, Aug. 1996, pp. 49-58.

[9] Z.S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 108-120, June 1997.

[10] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a texture cache architecture," *Proceedings of 1998 SIGGRAPH/Eurographics Workshop on Graphics Hardware,* pp. 133-142, Aug. 1998.

[11] H. Igehy, M. Elfridge, and P. Hanrahan, "Parallel texture cache," *Proceedings of 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 95-106, Aug. 1999.

[12] M. Woo, J. Neider, and T. Davis, *OpenGL programming guide*, Addison Wesley, 1996.

[13] R. Kempf and C. Frazier, *OpenGL reference manual*, Addison Wesley, 1996.