The Visual Computer

Two-level hierarchical Z-buffer with compression technique for 3D graphics hardware

Cheng-Hsien Chen,
Chen-Yi Lee

Dept. of Electronics Engineering, National Chiao Tung University, 1001, Ta Hsueh Road, Hsinchu, 300, Taiwan, R.O.C.
E-mail: chchen@royals.ee.nctu.edu.tw

Published online: 2 July 2003

The hierarchical Z-buffer is application-invisible and more efficient than the traditional Z-buffer for quickly rejecting hidden geometries. But there are construction and management issues associated with integrating a hierarchical Z-buffer into current graphics hardware. Here we present a two-level hierarchical Z-buffer algorithm, and provide solutions to these issues. Simulation results show that the bandwidth can be reduced by up to 35%. Moreover we propose a dynamic bi-level HZ-buffer compression technique that reduces the buffer size up by to 40%, and for which there is little performance degradation.

Key words: 3D graphics hardware – Hierarchical Z-buffer – Hierarchical Z-buffer compression

1 Introduction

Three-dimensional computer graphics are widely used in many applications. Large amounts of computational power, memory bandwidth, and database are consumed in the generation of lifelike images. Thus millions of transistors are put into the GPU (Graphics Process Unit) to boost performance. Although GPU processing power increases rapidly every year, the growth of bandwidth is not as fast as that of computing power. The bandwidth bottleneck prevents the GPU from performing at its full power. A block diagram of a traditional triangle-based 3D graphics rendering flow is shown in Fig. 1. There are several memory accesses in the pipeline. Texture mapping and the Z test (or depth test) consume most of the bandwidth [8]. Thus reducing memory accesses in such pipeline systems can enhance overall performance.

The Z-buffer test or depth test is the hidden surface removal stage. Most graphics hardware use the traditional Z-buffer algorithm [2] to resolve the visibility problem at the pixel level. Although the traditional Z-buffer algorithm is simple and easy for hardware implementation, its efficiency is not high and it does not make use of object-space coherence. Every pixel needs to access the Z-buffer, but most of them are invisible. It wastes memory bandwidth and computing power to transform, light, and rasterize those pixels.

A good visibility algorithm should quickly reject most of the hidden geometry in the scene. There are many speed-up techniques [7] for visibility tests and occlusion culling. Most of them need a lot of preprocessing before entering the hardware-rendering pipeline. Several examples can be found in the literature, such as object-space preprocessing with a binary-space partition (BSP) [3, 4], object-space octrees combined with an image-space Z pyramid [3, 4], portal culling [6], and image-space hierarchical occlusion maps [11]. However these techniques are not suitable for real-time interactive applications and cannot be integrated into existing hardware. A new hardware-support visibility-test algorithm is important for graphics system. The hierarchical Z-buffer visibility test can meet this requirement. The concept was first introduced by Greene [5] as the image-space Z-pyramid. ATI [8] was the first to integrate a hierarchical Z-buffer into a commercial product. nVidia [9] also integrated a hierarchical Z-buffer and a new crossbar memory management unit to relieve the memory bottleneck. But the issues associated with putting a hierarchical

The Visual Computer (2003) 19:467–479
Digital Object Identifier (DOI) 10.1007/s00371-003-0212-4

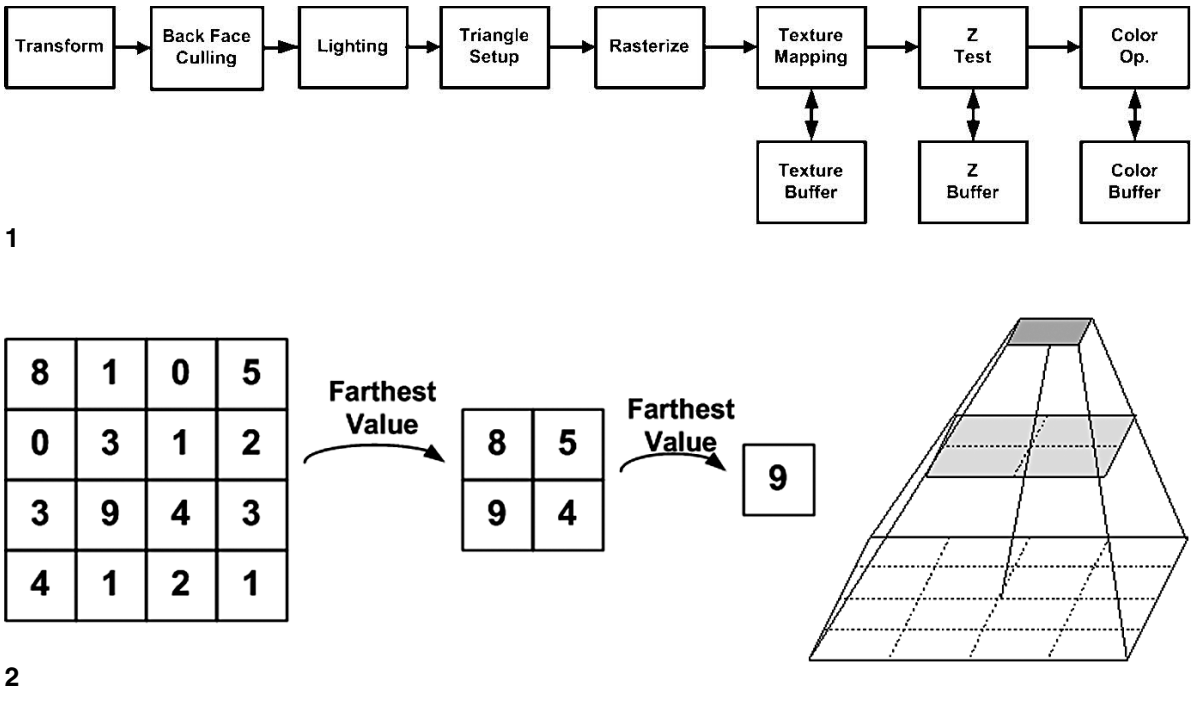


Fig. 1. Traditional triangle-based 3D graphics rendering flow
Fig. 2. 2×2 hierarchical Z-buffer

Z-buffer into hardware have not been discussed in literature.

A hierarchical Z-buffer (HZ-buffer) is a reduced resolution Z-buffer. Figure 2 shows a degree of 2 HZ-buffer. The original Z-buffer is the finest level of the pyramid. It combines four Z values at each level into one Z value at the next coarser level by choosing the farthest value inside the corresponding block. This process can be iterated to construct many levels of the hierarchy. Then the HZ-buffer can be used to determine the visibility of triangles and pixels, instead of querying the original Z-buffer.

In this paper, we present a two-level HZ-buffer visibility test algorithm, and solve the problem of bringing it into a hardware design. The issues of putting HZ-buffer into 3D hardware will be addressed in Sects. 2 and 3. Then we propose a compression technique to reduce the buffer size of the two-level HZ-buffer in Sect. 4. Simulation results are given in Sect. 5.

2 Previous work and issues of hierarchical Z-buffer hardware

The hierarchical Z-buffer visibility test was first proposed by Greene [5]. Greene's approach includes octree spatial subdivision to explore object-space and temporal coherence, while a Z pyramid is used to explore image-space coherence. Hierarchical scan conversion is incorporated into the HZ-buffer to solve the visibility problem. The concept of the Z pyramid becomes the HZ-buffer of today's hardware [8, 9]. When applying this technique to hardware, it is not easy to store, update, and manage so many levels of the HZ-buffer. Updating the lowest level of the hierarchy will incorporate possible updates of higher levels. The latency will be long for maintaining the HZ-buffer. Hierarchical scan conversion is also slower than traditional scan conversion. Thus the ATI HyperZ technique [8] uses just a one-level hierarchy. This hierarchy is constructed by 8×8-

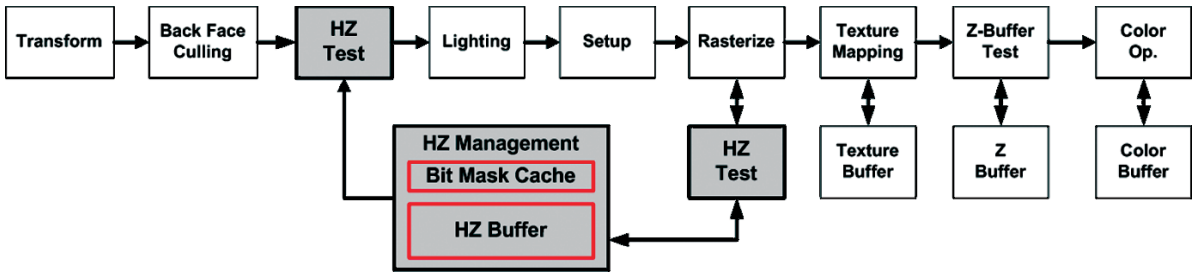


Fig. 3. Proposed two-level hierarchical Z-buffer visibility test architecture

or 4×4-block sizes from the original Z-buffer. Its HZ-buffer test stage is also behind the rasterization stage. The hardware can quickly reject several invisible pixels in one cycle. The visibility test is done at the pixel-level. It does not make use of the object-space coherence of the HZ-buffer at the triangle-level. In addition to triangle-based architectures, Xie and Shantz [10] proposed an adaptive hierarchical visibility algorithm for tiled architectures. The tiled architecture requires bucket sorting triangles in Z at each tile. It evaluates a cost function and pixel coverage when rendering primitives from near to far, and then construct the HZ-buffer at a selected time. This algorithm is well suited to the tiled architecture, but cannot be applied to current popular triangle-based architectures since they will not perform depth sorting for triangles.

There are two issues when bringing a HZ-buffer into hardware. The first is the HZ-buffer configuration. The configurations include the hierarchical block size, number of levels in the hierarchy, and depth numeric accuracy. There are trade-offs between hardware cost and performance. In order to shorten the latency of the visibility test, we suggest that the HZ-buffer should reside on the chip. This also simplifies the memory and HZ-buffer management. Thus the HZ-buffer should be of reasonable size for hardware implementation. The second issue for the HZ-buffer is how to update it. Taking a 8×8-block size one-level hierarchy for example, one value in the HZ-buffer represents the farthest depth in a 8×8 region of the original Z-buffer. It means that we have to fetch 64 depth values from the external Z-buffer and search the farthest one. This will introduce more memory access and operations, offsetting the benefit of the HZ-buffer. The algorithm that we present in this paper addresses all of the above issues. The HZ-buffer update problem is solved by adding a bit-mask cache. In addition, the HZ-buffer visibility tests are done both at the triangle-level and at the pixel-level in this algorithm. Finally, a HZ-buffer compression technique is presented to reduce the size of the HZ-buffer.

3 Two-level hierarchical Z-buffer

In this paper, we propose a HZ-buffer visibility test algorithm and discuss the problem of integrating it into hardware. The issues associated with the hardware HZ-buffer will be addressed in the following sections. Simulation results are shown in Sect. 5.

Figure 3 shows the block diagram of our two-level hierarchical Z-buffer (HZ-buffer) architecture. The difference from Fig. 1 is that we insert two HZ visibility tests. The first one after the transform stage is used to explore image-space coherence at the triangle-level and the second is at the pixel-level. The first HZ test can further reduce the loading and memory access of lighting, triangle-setup, and following operations by discarding invisible triangles. The second HZ test is done by testing the visibility of every pixel. The HZ-buffer management unit manages the HZ-buffer access and update procedure. A bit-mask cache is added to keep the temporal pixel coverage information to help updating of the HZ-buffer.

3.1 Hierarchical level selection

First we want to determine the levels of the hierarchy for the HZ-buffer. In a higher level of the hierarchy, the block covers a larger area than in a lower

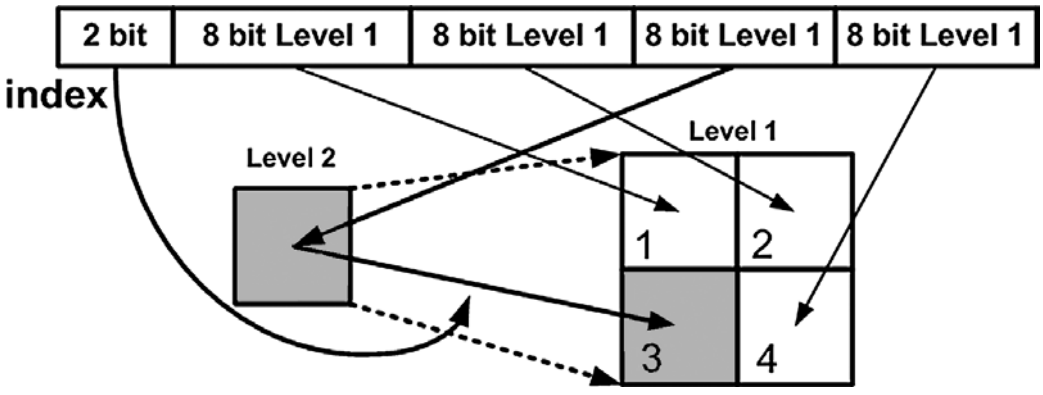


Fig. 4. Hierarchical Z-buffer format

level of the hierarchy. Thus higher hierarchy levels can get more image-space coherence at the triangle-level. This will discard more invisible triangles at the triangle-level HZ test. But the overall bandwidth reduction will be almost the same when combining the triangle-level and pixel-level HZ visibility tests. On the other hand, lower hierarchy levels can get more coherence at the pixel-level. Here we choose a two-level HZ-buffer configuration, because the size of the HZ-buffer is almost the same as for a one-level hierarchy. The complexity of HZ-buffer management is also smaller than for a higher hierarchy level HZ-buffer. Figure 4 is an example of the data format in the two-level HZ-buffer. There are four depth values for each block in level-1. Each depth value represents the farthest z-value in the region of the Z-buffer covered by one level-1 block. The farthest level-2 value is addressed by adding a 2-bit index in this format. Thus it does not need to allocate a dedicated memory for the second level of the hierarchy.

3.2 Hierarchical Z-buffer construction

The block size, depth numeric accuracy, and screen resolution will determine the size of the HZ-buffer. The formulation of the two-level HZ-buffer size is shown in (1). For high-level 16×16 and low-level 8×8 block sizes, the 8-bit accurate 16×16–8×8 HZ buffer is 21.7 KB under 1280×1024 screen resolution. The 8×8–4×4 HZ buffer is 87 KB. The buffer size should be chosen carefully for the trade-off between hardware cost and HZ-buffer efficiency.

We also simulate the performance for different levels of depth accuracy from 6-bit to 16-bit in Sect. 5. The results show that the performance difference is not significant for most cases. The difference is significant only for very high depth complexity scenes.

$$\begin{aligned} Buffer\ Size = {} & \frac{Width}{Low\_level\_Block_{width}} \times \frac{Height}{Low\_Level\_Block_{height}} \\ & \times Depth\_accuracy + \frac{Width}{High\_level\_Block_{width}} \\ & \times \frac{Height}{High\_Level\_Block_{height}} \times Bit_{index}, \end{aligned} \tag{1}$$

$$NB = \frac{High\_level\_Block_{width}}{Low\_level\_Block_{width}} \times \frac{High\_Level\_Block_{height}}{Low\_Level\_Block_{height}},$$

$$Bit_{index} = \lceil \log_2 NB \rceil .$$

3.3 Hierarchical Z-buffer visibility test

There are two HZ-buffer visibility tests in our algorithm: the triangle HZ visibility test and the pixel HZ visibility test. The triangle HZ test is done after the transform stage to discard invisible triangles. After a triangle passes the transform and back-face-culling stage, we first search the block in the level-2 HZ-buffer that fully covers the input triangle, as shown in Fig. 5a. If the triangle is fully covered, we test the farthest vertex of the triangle with the depth in the corresponding level-2 block. If the HZ test fails, we discard this triangle. A smaller triangle that fails the level-2 HZ test and is fully covered by one level-1 block (Fig. 5b) goes to the level-1 HZ test. If it fails the test, we discard it too. Those triangles that

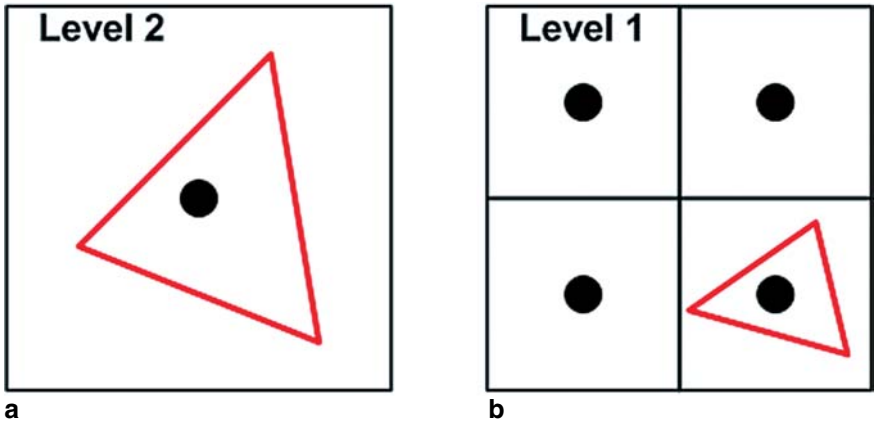


Fig. 5. Triangle HZ visibility test

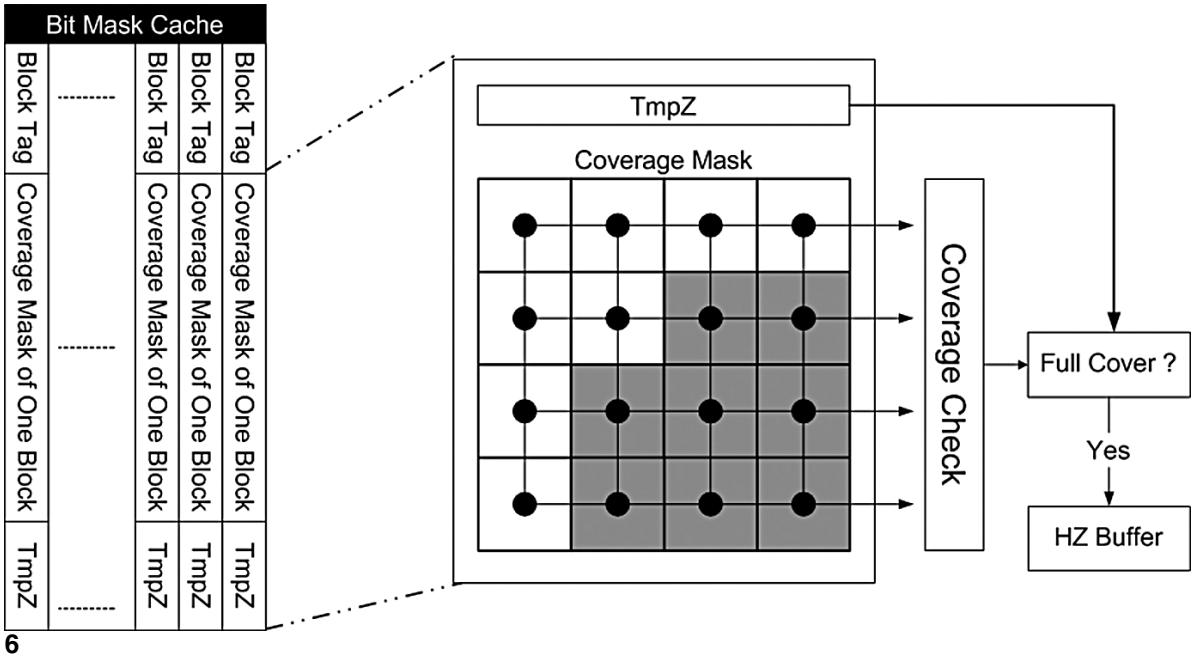
straddle multiple level-2 or level-1 blocks are left unchanged and pass the triangle HZ test. The second stage pixel HZ test will determine the visibility of these triangles.

The secondary visibility test is done at the pixel level after rasterization. When the triangles are scan-converted at the rasterization stage, the output pixels are compared with the corresponding low-level block depth. If the depth of the pixel is farther than the current low-level value, it means that the pixel is farther than any pixels in the low-level block and can be discarded. The pixel-level HZ test can be done for many pixels at once to enhance throughput. ATI [8] shows that it can discard 8 or 16 pixels at one cycle in the rasterization stage. By combining the triangle and pixel HZ tests, invisible triangles and pixels can be efficiently discarded.

3.4 Hierarchical Z-buffer management

HZ-buffer management is a problem for hardware implementation. The issues have not been mentioned in previous work [5, 8]. The challenge is how to update the HZ-buffer. The HZ-buffer is a reduced resolution of the Z-Buffer. Thus when the Z-Buffer is updated, the HZ-buffer should be updated too. The update process is to search the farthest z-value in one block. The block size may be 4×4, 8×8, or 16×16. Thus we have to fetch and compare 16, 64, or 256 z-values from the Z-buffer when updating the HZ-buffer. Comparing these z-values will increase the number of Z-buffer accesses and the latency. Thus the benefit of HZ visibility will be largely diminished. Here we propose a HZ-buffer management architecture to solve the updating problem.

The HZ management unit in Fig. 3 includes an on-chip HZ-buffer and a bit-mask cache. The HZ-buffer is used to store the two-level HZ depth. The format of the HZ-buffer is shown in Fig. 4. The bit-mask cache is shown in Fig. 6. The goal of the bit-mask cache is to cache the temporal pixel coverage information. It eliminates the process of reading the Z-buffer to determine the farthest depth in one block when the Z-buffer is updated. The bit-mask cache contains the block tag, the coverage mask of one level-1 block, and tmpZ for the temporally farthest z-value. The coverage mask and tmpZ are reset to zero at the beginning. When the pixel passes the HZ test and the Z-buffer test, both the Z-buffer and bit-mask cache should be updated. To update the bit-mask cache, first we search the block that covers this pixel in the bit-mask cache and set the coverage mask to '1' at the corresponding address. The z-value of this pixel is compared with tmpZ. If the new z-value is farther than tmpZ, tmpZ is updated with this z-value. Thus tmpZ always represents the farthest z-value of the pixel entering this block. When the bits in the coverage mask are all set to '1' in one block, it means that the block is fully covered by recently rasterized pixels, and then the HZ-buffer is updated with tmpZ. The coverage check is a simple bit-wise AND operation. When the block is fully covered, the cov-



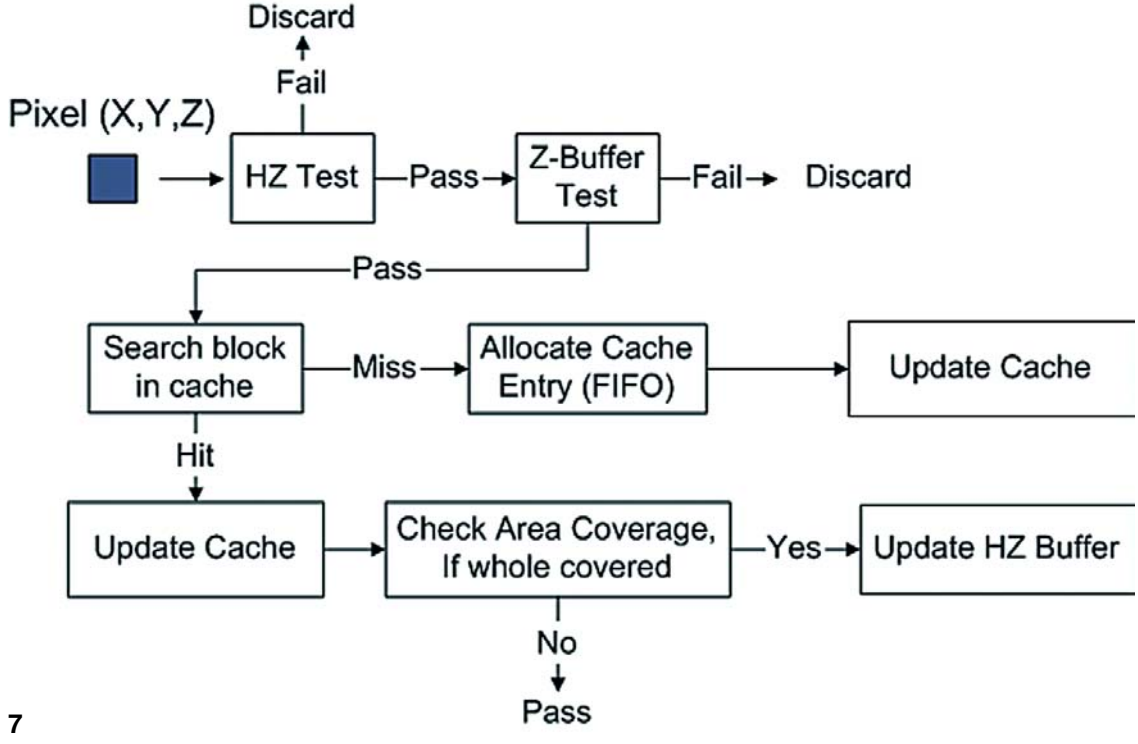


Fig. 6. Bit-mask cache organization
Fig. 7. HZ-buffer update flow

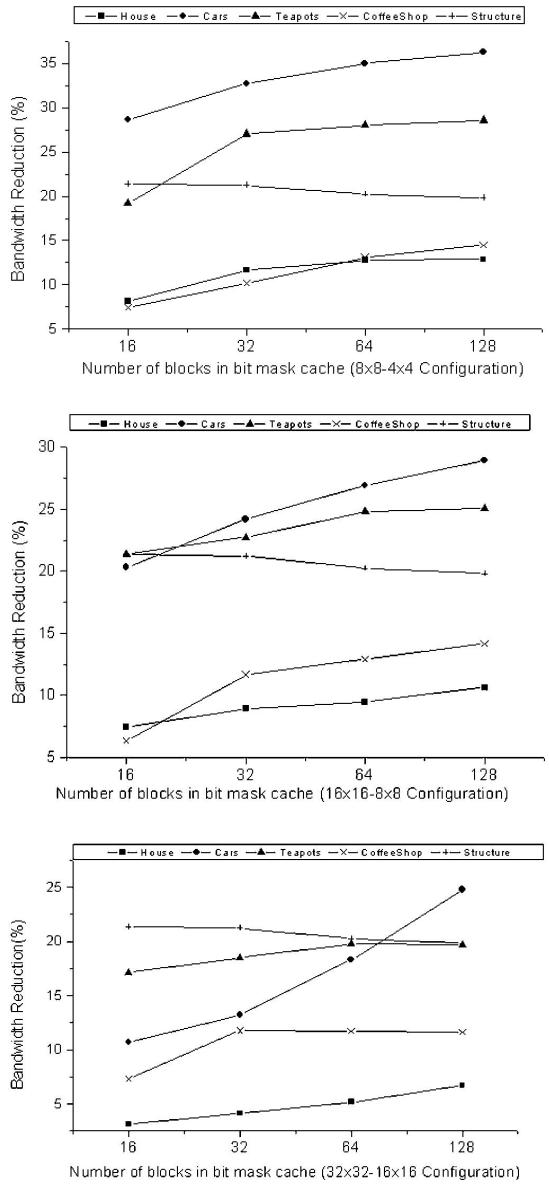


Fig. 8. Performance under different bit-mask cache sizes

erage mask and tmpZ are reset to zero for further processes.

By using the bit-mask cache, we do not need to fetch the Z-buffer and search the farthest z-value. This technique can make sure that the correct z-value is stored in the HZ-buffer and will not introduce error at the HZ test. Because the triangles entering the 3D pipeline are most likely to be located on nearby blocks, the bit-mask cache can get temporary coverage information very well. Since the block that is first touched will probably be covered first, the cache replacement mechanics of the bit-mask cache is FIFO (First In First Out). When the block is chosen for replacement, the contents of the coverage mask and tmpZ are discarded and reset. The cache size will also influence the efficiency of the HZ test. The performance under different cache sizes is also shown in Fig. 8. We can see that the cache size slightly influences the performance. For cache sizes of 16 and 128 blocks, the performance differences are within 10 percent in these simulations. Thus a small bit-mask cache of 16 or 32 blocks is sufficient for HZ management to cache temporal pixel coverage information. Figure 7 also shows the detailed flow for updating the HZ-buffer.

4 Dynamic bi-level hierarchical Z-buffer compression

The size of the HZ-buffer depends on the screen resolution, HZ-buffer configuration, and depth numeric accuracy. Table 1 shows the buffer size of a two-level HZ-buffer under 8-bit depth accuracy. Without compression, the original buffer size will consume a large chip area, especially for higher resolutions and for a small HZ block configuration. Since the contents of the HZ-buffer is changed dynamically at run-time, the compression technique should be simple and it should be easy to decode the bit stream at run-time. The spirit of the compression algorithm is not really to losslessly encode the bit stream of the HZ-buffer, but to preserve the correct depth order of triangles and pixels. The algorithm should not cause defects in the final image representation. Here we propose a compression technique, which is called dynamic bi-level HZ-buffer compression, to further reduce the HZ-buffer size.

From observation, we can see that the depth difference between one block and its neighbor blocks may not be large in the HZ-buffer. The reason is that nearby blocks are most likely to be within the same object at the same depth complexity. This feature is significant when the block size is small. Thus we can further make use of image-space coherence in the HZ-buffer to reduce the buffer size. It is trivial to select DPCM to encode the bocks. But DPCM will result in a variable length bit stream and increase the

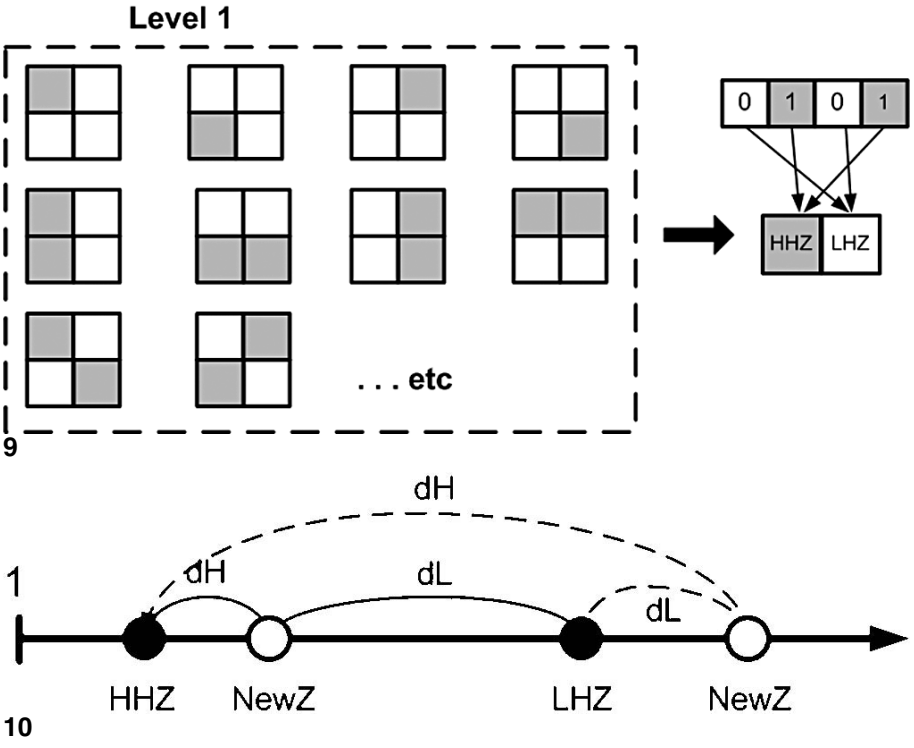


Fig. 9. The concept of dynamic bi-level HZ-buffer compression
Fig. 10. Relationship between NewZ, HHZ, and LHZ

complexity of HZ-buffer access. The performance of fixed-length DPCM is also not good for dynamic changes to the HZ-buffer at run-time. Figure 9 shows the concept of dynamic bi-level HZ-buffer compression. In Fig. 9, one level-2 block represents 2×2 down-sampling of level-1 blocks. Instead of keeping all of the depth of the four blocks, we classify these blocks into two groups and store only two depths. For those blocks that fall into a nearby depth region, we group them together and carefully assign the depth to these blocks. This procedure can be done by comparing the depth difference between the new depth and the existing depth (HHZ and LHZ) in corresponding blocks. Thus for one level-2 block, it can store depth complexity of two objects at any time. This is sufficient for most cases within one level-2 block. The final bit stream includes two z-values (HHZ, LHZ) and a 4-bit index. Each bit of the index indicates the depth of corresponding block, where '0' and '1' stand for HHZ and LHZ, respectively. When a new depth newZ enters, we may modify HHZ, LHZ and the index according to the position of newZ on the z-axis. When newZ is close to HHZ, we assign the block to HHZ. When newZ is close to LHZ, the operation depends on the relative distance of newZ and LHZ. Figure 10 shows the relationship of newZ, HHZ, and LHZ. The newZ will not be farther than HHZ and may fall on the left or right side of LHZ. The encoding process is to determine HHZ and LHZ, and preserve the correct depth order of these four blocks. The pseudo-code for the encoding process is shown below:

1. Initial:
$HHZ = 1.0$*;* $LHZ = 1.0$*;*
// Reset HHZ, LHZ to farthest value
$Index[0:3] =$ "1111"*;*
// Reset Index, '1' represents HHZ, '0' represents LHZ

2. Update:
// Input newZ and blockaddress (if blockaddress $= 2$ *here)*

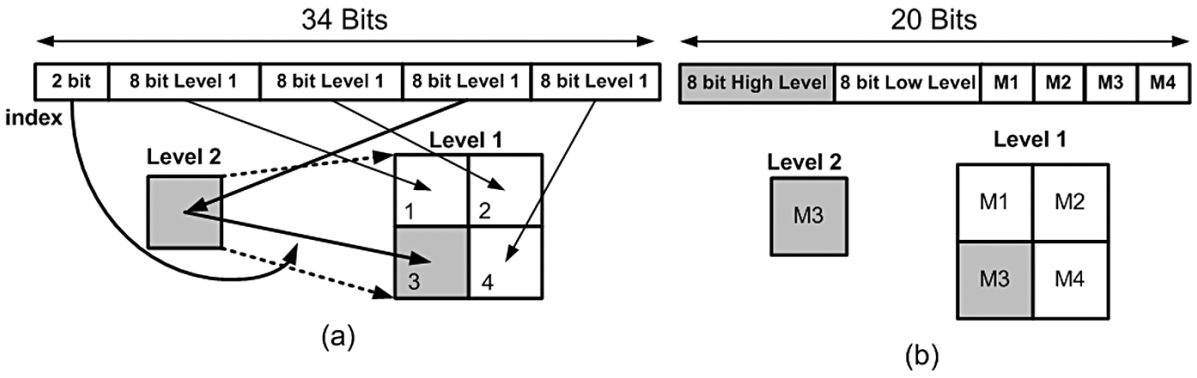


Fig. 11. Data format of **a** original HZ-buffer **b** dynamic bi-level Compression HZ-buffer

```
if(Index[0 : 3] == "1111")
   // Blocks are all HHZ
{   LHZ = newZ;
   // Store newZ as LHZ
    Index[blockaddress] = '0';
   // Set corresponding Index to '0' }
else
{   dH = |newZ − HHZ|;
    dL = |newZ − LHZ|;
    if(dH < dL)
   // newZ closer HHZ
    {   Index[blockaddress] = '1';
   // Set corresponding Index to '1'
   //assume blockaddress = 2
        No_other_block_belong_to_HHZ =
        ∼ (Index[0]|Index[1]|Index[3]);
        If(No_other_block_belong_to_HHZ)
        HHZ = newZ;
   // Else
   // Do nothing;       }
        else      // new Z closer LHZ
        {  Index[blockaddress] = '0';
           If(Index[0 : 3] = "0000")   //All block are LHZ
           {
            HHZ = Max(newZ, LHZ); // Update HHZ
            LHZ=HHZ; // Reset HHZ=LHZ
            Index[0 : 3] = '1111';       // Reset Index }
        Else
        { LHZ = Max(newZ, LHZ); }   // Update LHZ
          }
}
```

For next newZ, go to step2

During the encoding process, the contents of the block are changed dynamically and the correct depth order remains unchanged. The bit stream will be reduced from 34- to 20-bit, as shown in Fig. 11. The size of the HZ-buffer will be decreased by about 40%, as shown in Table 1. Table 3 shows the result under dynamic bi-level compression. The performance is only degraded by several percent due to compression. Thus the dynamic bi-level compression algorithm can not only reduce the buffer size, but can also preserve the performance.

5 Simulation results and analysis

Here we choose four scenes in Fig. 12 to simulate the efficiency of our two-level HZ-buffer algorithm. Our simulation is implemented with a standard graphics

Table 1. HZ-buffer size (in KB) with and without compression

	32×32–16×16		16×16–8×8		8×8–4×4	
Compression	No	Yes	No	Yes	No	Yes
1600×1200	7.97	4.69	31.88	18.75	127.5	75
1280×1024	5.44	3.2	21.76	12.8	87	51.2
1024×768	3.26	1.92	13.06	7.68	52.2	30.7
800×600	2	1.17	7.97	4.69	31.9	18.75

Table 2. Percentage of bandwidth reduction with different levels of depth numeric accuracy (1280 × 1024, 64 blocks bit mask cache size)

		House	Car	Teapot	Coffee Shop	Structure
8 × 8–4 × 4	6	12.59	32.92	27.67	11.91	34.73
	8	12.77	34.99	28.02	13.08	36.79
	12	12.86	35.55	28.08	13.20	37.41
	16	12.86	35.61	28.08	13.24	37.49
16 × 16–8 × 8	6	9.41	25.60	24.56	12.15	18.4
	8	9.44	26.91	24.80	12.90	20.24
	12	9.45	27.41	24.92	13.23	20.82
	16	9.45	27.49	24.92	13.25	20.85
32 × 32–16 × 16	6	5.14	17.43	19.71	11.29	3.53
	8	5.16	18.30	19.78	11.73	3.87
	12	5.18	18.79	19.83	12.12	3.99
	16	5.18	18.84	19.84	12.13	4.04

Table 3. Triangles, pixels, and bandwidth reduction percentage under various HZ configurations and compression (64 blocks bit-mask cache, 1280 × 1024 resolution)

		House		Cars		Teapots	
Total Triangles		316 671		454 746		102 400	
Memory Access		436 234		387 337		147 354	
Dynamic Bi-level Compression		No	Yes	No	Yes	No	Yes
		%	%	%	%	%	%
8 × 8–4 × 4	*Triangle*	20.7	21.76	34.9	39.39	5.5	9.9
	Pixel	17.2	16.48	44.8	34.66	39.6	33.82
	Bandwidth	12.8	12.27	35	28.78	28	26.72
16×16–8×8	*Triangle*	23.6	21.19	34.6	35.73	12.3	14.62
	Pixel	10.2	9.95	30	23.02	28.7	21
	Bandwidth	9.4	8.68	27	22.63	24.8	23.29
32×32–16×16	*Triangle*	14.9	17.22	32.5	30.53	14.4	13.61
	Pixel	4.67	5.13	14.5	8.56	14.8	10.36
	Bandwidth	5.2	5.48	18.3	21.03	19.8	18.39

pipeline in C++. Table 2 shows the performance for different levels of z-value accuracy. Table 3 shows the simulation results for different HZ-buffer configurations and compression at 8-bit HZ-buffer depth accuracy for a 64-block bit-mask cache size with a 1280 × 1024 image size. The complexity of the test images is within 100∼600 K triangles. Back-face culling [1] is enabled in the simulation. Without back-face culling, the HZ-buffer can also take care of the visibility of the back-face very well. The Z-buffer memory access times are listed in Table 3. The numbers shown in Table 3 are the percentage of triangles that fail the first triangle-level HZ test, the percentage of pixels that fail the secondary pixel-level HZ test, and the overall memory access reduction.

From Table 2, we can see that the performance does not change to a great extent with different levels of depth numeric accuracy. This is because for an 8-bit depth, the HZ-buffer can represent at most 256-level depth complexity, and this is sufficient for most API's. The performance difference will be large when the depth complexity of the scene is very large. Thus for hardware implementation, 8-bit HZ accuracy is fine for most applications and memory alignment considerations. Higher accuracy will increase the efficiency. There is a tradeoff between performance and buffer size. From Table 3, we can see that the overall bandwidth reduction is 10∼35 percent. The dynamic bi-level compression technique reduces the performance by only several percent. The simulation results are application dependent. The properties of different applications may affect the results significantly. For high-occlusion models (Cars, Teapots, and Structure), the HZ-buffer efficiency is high. For a low-depth complexity image (House), the efficiency is low. Different HZ block-size configurations also result in different efficiencies of the HZ test. For a large triangle image (Teapots), the efficiency of the HZ test at the triangle level is low for a small block HZ configuration

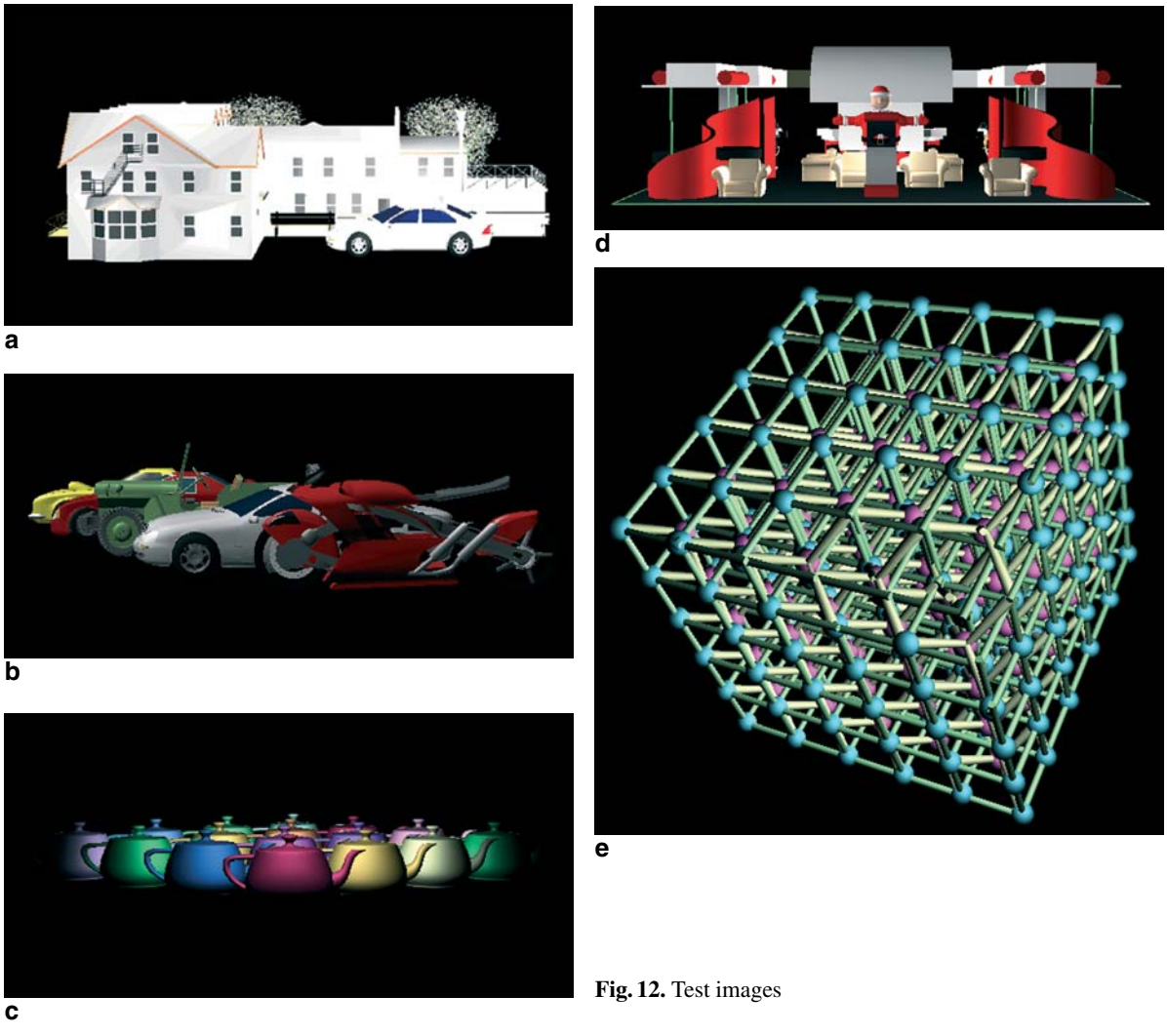


Fig. 12. Test images

(8×8–4×4). Since a larger block size can give more triangle-level coherence in image space, increasing the HZ block size to 32×32–16×16 will increase the efficiency. For a small triangle image (Structure), the performance increases largely due to good pixel-level coherence for a small block size. The second pixel-level HZ test efficiency is reduced by increasing the block size. Obviously, a small block size will give a more detailed z-value and result in a more accurate HZ test and image-space coherence. But a small block size will increase the HZ buffer size. A large block size will also result in greater effort in updating the HZ-buffer. Thus the HZ-buffer configuration is a trade-off between performance and hardware cost. Here we suggest a 16×16–8×8, 8-bit accuracy HZ-buffer configuration. A 32- or 64-block bit-mask cache size is sufficient for most applications.

6 Conclusion

Fast visibility tests and occlusion-culling algorithms have been important topics in 3D graphics over the past decades. However most of the solutions are not easy to implement on current hardware. A fast

and efficient hardware-support visibility-test algorithm is very important. The HZ-buffer meets this requirement. It is a good approach for accelerating visibility tests and efficiency. Since it is invisible in applications, modification of the API is not required to get the benefit of the HZ-buffer. In this paper, we propose a two-level HZ-buffer algorithm, and solve the problems of hardware implementation. The visibility tests are done both at the triangle-level and at the pixel-level. We also show how to construct and manage the HZ-buffer when applying it to hardware. A bit-mask cache is proposed to store the temporal pixel coverage information and solve the update problem of the HZ-buffer. Simulation results show that a small bit-mask cache can achieve good performance and solve the update problem. We also propose a dynamic bi-level HZ-buffer compression technique to further reduce the HZ-buffer size while maintaining good performance. The compression technique is easy to encode and decode. It can reduce the HZ-buffer by about 40% of its original size.

For hardware implementation considerations, we suggest a 16×16–8×8 HZ-buffer configuration, 8-bit z-value accuracy, and a 64-block bit-mask cache size. This configuration performs well for most cases. The bandwidth reduction under different HZ-buffer configurations is also shown for the trade-off between performance and hardware cost considerations. By applying our technique to hardware, the invisible triangles and pixels can be quickly discarded. The overall graphics pipeline can also run smoothly with the overhead of HZ-buffer hardware.

For future content-rich applications, the hardware should integrate more advanced hidden surface removal technology, such as object-space occlusion and temporal-space coherence. The graphics library should also support these techniques and be incorporated into the hardware to provide a high-efficiency low-overhead solution.

Acknowledgements. This work was supported by the National Science Council of Taiwan, ROC, under Grant No. NSC90-2218-E-009-080.

References

1. Blinn J (1996) Jim Blinn's corner: a trip down the graphics pipeline. Morgan Kaufmann Publishers, San Francisco, pp 191–197
2. Catmull E (1975) Computer display of curved surfaces. In: Proc IEEE Conf Computer Graphics, Pattern Recognition and Data Structures, pp 11–17
3. Fuchs H, Kedem ZM, Naylor BF (1980) On visible surface generation by a priori tree structures. Comput Graphics 14(3):124–133
4. Gordon D, Chen S-H (1991) Front-to-back display of BSP trees. IEEE Comput Graphics Appl 11(5):79–85
5. Greene N, Kass M, Miller G (1993) Hierarchical Z-buffer visibility. In: Proc SIGGRAPH '93, pp 231–238
6. Luebke D, Georges C (1995) Portals and mirrors: simple, fast evaluation of potentially visible sets. In: Proc 1995 Symp Interactive 3D graphics, pp 105–106
7. Moller T, Haines E (1999) Real-time rendering. A.K. Peters, Natick, pp 192–218
8. Morein S (2000) ATI Radeon HyperZ technology. In: Eurographics Hardware Workshop 2000, Hot3D Panel
9. nVidia (2001) Technical brief: Geforce3: Lightspeed memory architecture. http://www.nvidia.com
10. Xie F, Shantz M (1999) Adaptive hierarchical visibility in a tiled architecture. In: Proc 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware, pp 75–142
11. Zhang H, Manocha D, Hudson T, Hoff KE (1997) Visibility culling using hierarchical occlusion maps. In: Proc 24th Conf Computer Graphics and Interactive Techniques 1997, pp 77–88

Photographs of the authors and their biographies are given on the next page.

CHEN-YI LEE received his B.S. degree from the National Chiao Tung University, Hsinchu, Taiwan, in 1982, and his M.S. and Ph.D. degrees from the Katholieke University Leuven (KUL), Belgium, in 1986 and 1990, respectively, all in electrical engineering. From 1986 to 1990 he was with IMEC/VSDM, working in the area of architecture synthesis for DSP. In February 1991, he joined the faculty of the Electronics Engineering Department, National Chiao Tung University, Hsinchu, Taiwan, where he is currently a Professor. His research interests mainly include VLSI algorithms and architectures for high-throughput DSP applications. He is also active in various aspects of high-speed networking, system-on-chip design technology, very low bit rate coding, and multimedia signal processing.

CHENG-HSIEN CHEN was born in Tainan City, Taiwan, R.O.C., on August 6, 1975. He received his Ph.D degree from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in 2003. His research interests include VLSI algorithms and architectures (include 3D graphics and video systems), and memory optimization for system-on-chip design.