Rendering of 3D Wavelet Compressed Concentric Mosaic Scenery with Progressive Inverse Wavelet Synthesis (PIWS)

Yunnan Wu, Lin Luo[†*], Jin Li and Ya-Qin Zhang[‡]

[†]University of Science and Technology of China, Hefei, 230026, China
[‡]Microsoft Research China, 49 Zhichun Road, Haidian, Beijing 100080, China

ABSTRACT

The concentric mosaics offer a quick solution to the construction and navigation of a virtual environment. To reduce the vast data amount of the concentric mosaics, a compression scheme based on 3D wavelet transform has been proposed in a previous paper. In this work, we investigate the efficient implementation of the renderer. It is preferable not to expand the compressed bitstream as a whole, so that the memory consumption of the renderer can be reduced. Instead, only the data necessary to render the current view are accessed and decoded. The progressive inverse wavelet synthesis (PIWS) algorithm is proposed to provide the random data access and to reduce the calculation for the data access requests to a minimum. A mixed cache is used in PIWS, where the entropy decoded wavelet coefficient, intermediate result of lifting and fully synthesized pixel are all stored at the same memory unit because of the in-place calculation property of the lifting implementation. PIWS operates with a finite state machine, where each memory unit is attached with a state to indicate what type of content is currently stored. The computation saving achieved by PIWS is demonstrated with extensive experiment results.

Keywords: Wavelet compression, synthesis, lifting, progressive inverse wavelet synthesis (PIWS), concentric mosaics, random data access, selective decompression

1. INTRODUCTION

The concentric mosaic [1] has proven itself to be a very useful tool for generating real-time photo realistic views of the synthetic and real world scenery. By rotating a single off-center camera and recording the captured images at regular intervals, a concentric mosaic scenery is built up quickly, and novel views can be easily obtained by interpolating existing light rays. Under the paradigm of image-based rendering (IBR), concentric mosaic gives a 3D parameterization of the plenoptic function. The concentric mosaic greatly eases the task of 3D scene acquisition and navigation. However, the data amount is huge, which presents a heavy burden for storage, transmission and display. Efficient compression algorithm is thus indispensable for the application of the concentric mosaics.

In a previous work [2], we have proposed a compression scheme based on 3D wavelet transform. 3D Wavelet is efficient to compact the energy and to exploit the redundancy within frame and across frames. The multi-resolution structure of the 3D wavelet is also highly desirable, especially in the Internet streaming application where a single compressed bitstream need to satisfy a variety of display resolution and quality settings. However, issues of real time decoding and rendering are not addressed in [2]. It is assumed that the entire compressed concentric mosaic bitstream is fully decoded prior to the rendering. Fully decompressing the concentric mosaics requires not only a huge memory at the renderer to hold the entire expanded scene, but also a long initial delay. This approach is not a satisfactory solution, because at any time of rendering, only a view of the concentric mosaic is rendered, which accesses only a small portion of the data set. In this work, our objective

[*] The work was performed when Mr. Yunnan Wu and Ms. Lin Luo were interns at Microsoft Research China.
Correspondence: Email: jinl@microsoft.com. Telephone: (86-10) 6261-7711 Ext. 5793. Fax: (86-10) 6255-5337.

is to develop a data access mechanism so that a portion of the 3D wavelet compressed data can be randomly accessed and decoded with high efficiency.

Most wavelet decompression techniques today assume an entire image and/or a block of data is accessed and decoded. There are extensive researches on speeding up of the wavelet encoding and decoding system for an image frame, e.g. [7][8]. In volume graphics, decoding cubes of data is also investigated in [3][4][5][6]. To enable easy access of partial data, short kernel wavelet filters such as the Haar filter are often adopted. However, Haar filter is not efficient in energy compaction, and the blocking artifact introduced by the Haar filter is very objectionable visually.

In this paper, we present the progressive inverse wavelet synthesis (PIWS) scheme that can access and partially decode wavelet compressed data without speed penalty. The key idea is to develop a mixed cache, where each memory unit is associated with a state machine that transits among wavelet coefficient, intermediate lifting value and output pixel. With the mixed cache, the intermediate lifting values are not dropped, therefore, PIWS guarantees a minimum amount of computation for a randomly accessed data set. PIWS is designed specifically in this paper for just-in-time (JIT) rendering of the concentric mosaics, i.e., to perform only the operation necessary to render the current view. However, it may also be extended to other wavelet compressed dataset, such as wavelet compressed volume graphics or 3D texture. PIWS may be applied to any wavelet filter with a lifting implementation.

This paper is organized as following. We briefly review the capturing, compression, decompression and rendering system of the 3D wavelet compressed concentric mosaics in section 2. In Section 3, accessing and selective decoding a portion of data with the PIWS algorithm is described in detail. Experimental results are shown in section 4. Finally, conclusions are given in section 5.

2. THE CONCENTRIC MOSAICS AND 3D WAVELET COMPRESSION

A concentric mosaic scene is captured by mounting a camera at the end of a leveled beam, and shooting images at regular intervals as the beam rotates. The resulting dataset is a shot sequence, which can be denoted as $f(n,w,h)$, where n indexes the camera shot, w and h represent the horizontal and vertical position of each ray within a shot, respectively. Let N be the total number of shots during the 360 degrees rotation, W and H be the width and the height of each shot. N is often set between 900 and 1500, which means that the camera shots are very dense, typically 2.5 to 4 shots per degree. Alternatively, the entire data volume can be interpreted as a stack of panorama images co-centered with the camera track, where each panorama is the mosaic image of vertical slits with the same w. The concentric mosaic is named after such a data structure.

Rendering of the concentric mosaics involves reassembling slits from existing shots. Shown in Figure 1, let P be a novel viewpoint and AB be the virtual field of view to be rendered. We split the view into multiple vertical slits, and render each of them independently. For example, when the slit PV is rendered, we simply search for the slit $P'V$ in the captured dataset, where P' is the intersection between ray PV and the camera track, as they are just the same because the intensity of the ray does not change along a straight line. Because of the discrete sampling, the exact slit $P'V$ might not be found in the captured dataset. Let the four sampled slits closest to $P'V$ be P_1V_{11}, P_1V_{12}, P_2V_{21} and P_2V_{22}, where P_1 and P_2 are the two nearest captured shots on the two sides of the intersection point P' along the camera track, P_1V_{11} and P_1V_{12} are slits just beside P_1V in shot P_1, and P_2V_{21} and P_2V_{22} are slits beside P_2V in shot P_2. We may bilinearly interpolate the four slits to approximate the content of $P'V$ (denoted as bilinear interpolation mode), or, possibly due to complexity and network bandwidth constraint, we may use the one slit closest to $P'V$ to represent it (denoted as point sampling mode). It is obvious that the bilinear interpolation mode results in a better rendering quality of the scene, at the price of a slower rendering speed. In either case, the content of the slit $P'V$ is recovered, which is then used to render slit PV in the rendered view. The environmental depth information may be helpful to find the best approximating slits and alleviate the vertical distortion. More detailed description of concentric mosaics rendering may be found in [1].

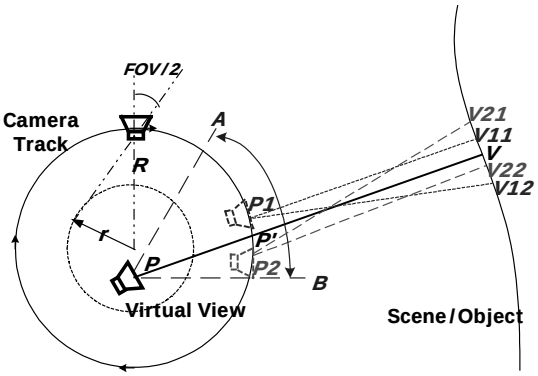

Figure 1 Rendering with the concentric mosaics

A 3D wavelet compression algorithm for the concentric mosaics is proposed in [2], the system architecture can be shown in Figure 2. The initial captured shot sequence first flows through a panorama alignment module to enhance its cross mosaic correlation, where each mosaic is circularly shifted. A 3D wavelet decomposition is then applied to the aligned mosaics, concentrating the source energy into relatively few large coefficients. After that, wavelet coefficients in each subband are split into individual blocks, scalar quantized and entropy encoded into an embedded bitstream. Finally, a rate-distortion optimized bitstream assembler truncates and concatenates the block bitstreams to form the compressed bitstream.

The rendering and decompression operations are procedurally opposite of the capturing and compression operation. In [2], block bitstream is first parsed from the compressed bitstream. Each wavelet coefficient block is then entropy decoded and inverse quantized. After that the inverse wavelet lifting and inverse alignment is performed. The fully expanded concentric mosaic data set is then used for the rendering. The approach is straightforward, however, a large memory is necessary to hold the decompressed concentric mosaics, and there is a long initial delay to decode the entire data set.

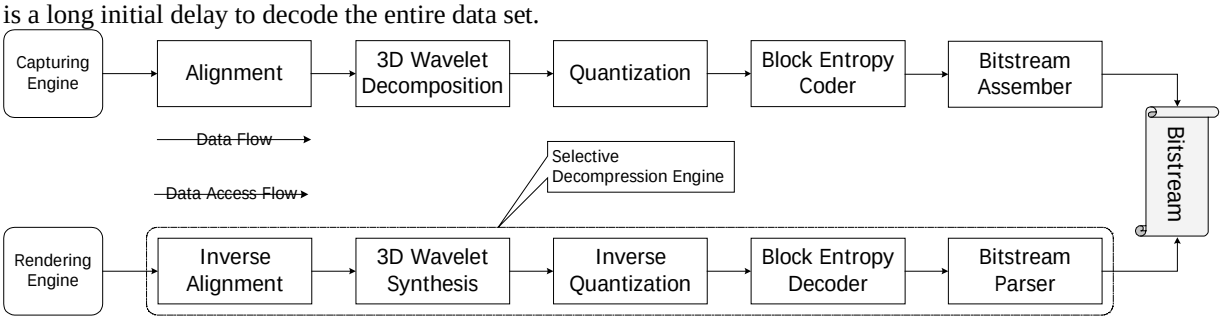

Figure 2 The 3D wavelet compression/selective decompression system architecture

To render any view of the concentric mosaics, only a small set of vertical slits are accessed. We therefore implement the function of just-in-time (JIT) rendering, i.e., to perform only the operation necessary to render the current view. Driven by the rendering engine, JIT renderer issues request for a set of slits needed for the current view to the selective decompression engine. The wavelet coefficients necessary to decode the slits are located, and inverse lifted to recover the accessed slits. When necessary, blocks of wavelet coefficients are entropy decoded and inverse quantized. There are previous works on selective decoding. Given the point or region of interest, one can derive the perfect reconstruction mask using the wavelet filter information and perform the synthesis work, as shown in [10]. The wavelet synthesis module has been found to be the bottleneck in the decompression system. The progressive inverse wavelet synthesis (PIWS) is proposed so that a minimum computation is performed to recover the accessed slits. Consequently, the decompression engine is accelerated to a large extent.

3. PROGRESSIVE INVERSE WAVELET SYNTHESIS (PIWS)

Progressive inverse wavelet synthesis (PIWS) is based on the inverse lifting operation. The core is to provide a minimum number of computations for the recovery of a few randomly distributed data points. We first review the forward and inverse lifting operation in Section 3.1. The one dimensional PIWS algorithm is investigated in Section 3.2. Finally, the PIWS for concentric mosaic rendering is explored in Section 3.3.

3.1 Lifting Scheme

Lifting is a memory and computation efficient implementation of the forward wavelet analysis and inverse wavelet synthesis. Every FIR wavelet filter can be factored into lifting steps[12]. A sample forward and backward one-dimension biorthogonal 9-7 lifting wavelet is illustrated in Figure 3 and Figure 4, respectively. In Figure 3, the original data x_0, x_1,$\cdots$,x_8 are input at the left, while the decomposed wavelet coefficients are output at the right. The high pass and low pass wavelet coefficients are interleaved at the output. It is observed that the wavelet coefficients are calculated through four stages of computation, each of which involves only half the nodes. An elementary forward lifting unit is shown in the right of Figure 3. An important feature of lifting is the in-place execution. When a calculation is executed in the elementary unit, the previous node is never used in subsequent stages, thus the memory is reused to store the resultant value. Through lifting, no additional memory is needed in the intermediate steps. The memory unit can be used to store the original data, the intermediate lifting result and the final wavelet coefficient.

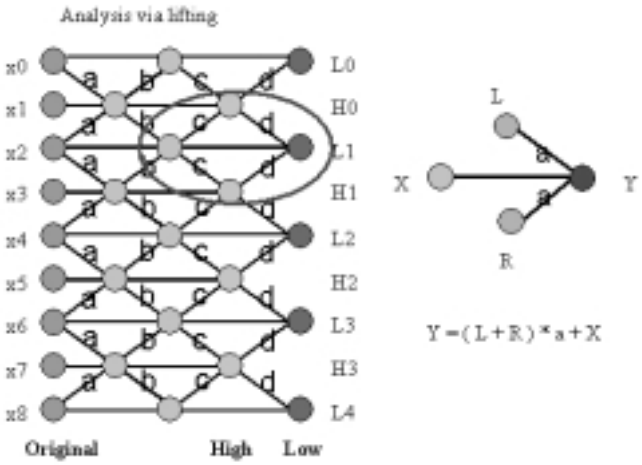

Figure 3 Forward wavelet analysis via lifting and an elementary forward lifting unit.

Because each elementary forward lifting unit can be straightforward inversed to an inverse lifting unit, the inverse wavelet synthesis can be easily derived by directly inverting the data flow of the forward lifting, as shown in Figure 4. The same in-place execution property also holds for the inverse lifting operation.

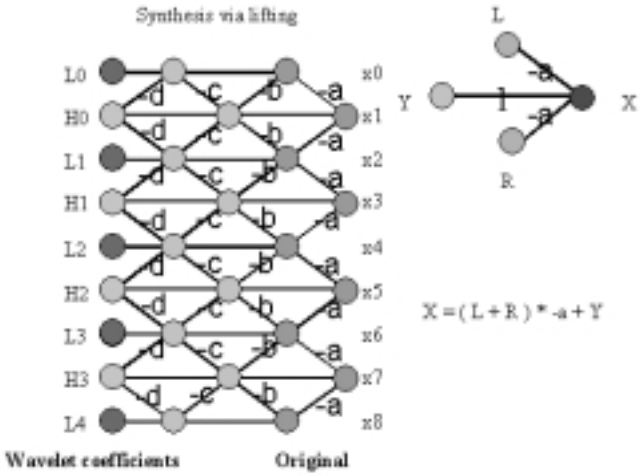

Figure 4 Inverse wavelet synthesis via lifting and the elementary inverse lifting unit.

3.2 1D data access through progressive inverse wavelet synthesis

Let us first consider the access of data points in one dimension. We use the biorthogonal 9-7 wavelet filter for explanation. However, other wavelet filters can be easily applied as well. The data is recovered through inverse lifting, as shown in Figure 4. Each data at odd index is recovered from 9 wavelet coefficients, with 5 high pass coefficients and 4 low pass coefficients. Each data at even index is recovered from 7 wavelet coefficients, with 4 high pass coefficients and 3 low pass coefficients. Let the coefficients necessary to decode a data point be its covering coefficients. If a long segment of data is recovered with lifting, on average 4 additions and 2 multiplications are needed to recover each data point. However, to recover a single data point at odd index, 20 additions and 10 multiplications are necessary. The computation load is 12 additions and 6 multiplications for the recovery of a single data point at even index. On average, 16 additions and 8 multiplications are needed if each data point is synthesized separately.

Suppose a set of data points is to be randomly accessed from the compressed data set. We may decode each data point independently, i.e., decode the wavelet coefficients covering the data point and then perform the inverse lifting. There is a heavy computation burden with the point decoding approach, as on average 4 times more calculation is needed if we decode each data point separately. An alternative approach is to first decompress all the data and then provide random access. Though the average computation load is low when the entire data set is decompressed, there is a long initial delay if only a few data points are accessed. Moreover, a large memory buffer is also needed to hold the expanded data set.

In this work, we propose the progressive inverse wavelet synthesis (PIWS) for the random data access. PIWS achieves a minimum computation for the recovery of a random data set. Let $y_0, y_1, \cdots, y_{N-1}$ be a set of transform coefficients, where low and high pass coefficients are interleaved at even and odd index, respectively. Let the recovered data be denoted as $x_0, x_1, \cdots, x_{N-1}$. In the lifting operation, both the transform coefficients and the recovered data can be placed at the same memory location, based on the in-place execution property of lifting. Let the memory unit which holds the coefficient y_i and the data x_i be denoted as m_i. A state s_i is attached to the memory unit m_i. We note that for the 9-7 biorthogonal wavelet, two lifting operations are performed for each memory unit before the data is recovered. Therefore, there are 3 possible states for each memory unit, as shown in Table 1. The PIWS operates as a state machine, at first, all the memory unit are in state 0, and after the accessing operation, the accessed data are brought to state 2.

Table 1 States for 1D progressive inserve wavelet synthesis

State	Information
0	Wavelet coefficient
1	Intermediate result after 1st lifting
2	Recovered data

Each transition of state in PIWS is driven by one elementary lifting operation. All the possible state transition of the lifting can be shown in Table 2. The lifting operation at even index is different from that at the odd index, so there are four different state transition operations.

Table 2 State transition operations for 1D inverse lifting.

Original State					Destination State
I	m_i	m_{i-1}	m_{i+1}		m_i
Even	0	0	0		1
Odd	0	1	1		1
Even	1	1	1		2
Odd	1	2	2		2

The access of data point x_i is equivalent to bring the memory unit m_i to destination state 2, which can be achieved through an elementary inverse lifting operation if unit m_i is in state 1, and its neighborhood units m_{i-1} and m_{i+1} are in state 2 (for odd i) or 1 (for even i). We may implement the access of data point with a recursive function *access(i, state)*, where i is the accessed data point, and the *state* is 2 for data access, 1 and 0 for intermediate execution and wavelet coefficients, respectively. The pseudo code can be written as Figure 5.

```
access (i, state)
{
   if (s_i <state)
   {
     if (i is even) {access (i-1,state-1);access(i+1,state-1); access(i,state-1); }
            else  {access (i-1,state);  access(i+1,state);  access(i,state-1); }
     perform an elementary lifting;
   }
   return m_i
}
```

Figure 5 Pseudo code for 1D progressive inverse wavelet synthesis

At first, the contents in all memory units are wavelet coefficients at state 0. As data are accessed, the state of the memory unit gradually transits to higher states. For any randomly accessed data points, only the operation necessary to calculate the accessed data point is performed, which greatly reduces the computation load.

3.3 Slit access through progressive inverse wavelet synthesis in the concentric mosaics

We first investigate the single scale PIWS algorithm for the 3D wavelet transform, and then extend to the case of multiple scales. Let the axis along the slit be the slit axis, and the other two axes be the horizontal and vertical axes, respectively. Since in the concentric mosaics, data are always accessed by slits, the memory unit is set as

one slit, instead of each individual data point. A state is again assigned to each memory unit. A mixed cache is again used to hold the wavelet coefficients, intermediate lifting results and the recovered pixels all at one place. Let the memory unit be denoted as $m_{i,j}$, and its state as $s_{i,j}$. Similar to the 1D PIWS case, we index the memory unit with the recovered slit, and interleave low and high pass coefficients at even and odd index, both for the horizontal and vertical axes. It is assumed that the horizontal inverse wavelet lifting is performed first and then the vertical lifting follows. The slit inverse wavelet synthesis is performed at the last. However, since the wavelet transform is separable, other transform orders are plausible as well. A list of feasible states can be shown in Table 3.

Table 3 States of the 3D progressive inverse wavelet synthesis

No.	State	Information
0	n	No coefficients available
1	x0	Entire slit decoded
2	x1	1st stage horizontal lifting performed
3	x2=y0	2nd stage horizontal lifting performed
4	y1	1st stage vertical lifting performed
5	y2	2nd stage vertical lifting performed
6	z	Slit lifting performed, data recovered.

The state goes through a single direction transition with n→x0→x1→y0→y1→y2→z. PIWS first occupies the memory unit with the wavelet coefficient, and then the intermediate horizontal lifting result, after that the intermediate vertical lifting result, and finally the recovered pixel, all in the same place. A similar recursive algorithm as the one in Figure 5 is developed for the PIWS slit access. We illustrate in Figure 6 the states of the cache when a single slit at even horizontal and odd vertical index is accessed. The covering area of the slit covers 7x9 slits. The access of a single slit is computational expensive. In fact, it takes 66 multiplications and 132 additions per sample to access the slits in Figure 6. However, the intermediate lifting results are not dropped, they can be used when neighborhood slits are accessed for significant computation savings. Although accessing a single slit, the total operational cost of the PIWS is the same as straightforward synthesizing the slit, none of the intermediate results is wasted. Ultimately, as more and more slits are accessed, the computational complexity of PIWS approaches that of the full inverse wavelet synthesis, i.e., 6 multiplications and 12 additions per sample per scale.

The memory requirement can be greatly reduced for the PIWS enabled concentric mosaic renderer too. With PIWS, we do not need to expand the compressed concentric mosaic scene. Instead, a cache is allocated to hold the most recent used memory units of PIWS, whether they are the wavelet coefficients, the intermediate lifting results or the recovered pixels. No memory is allocated for slits in state n, as such slits hold no data. Whenever the cache is used up, the less frequently used slits are swapped out and pushed back to state n, whose memory is then reused for the newly accessed slits.

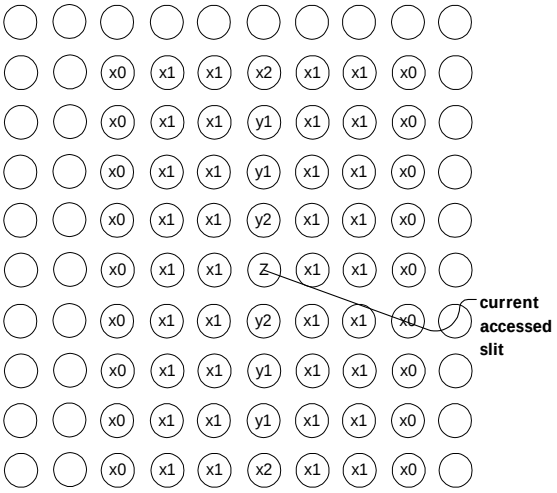

Figure 6 Cache states when a single slit is accessed

In the concentric mosaics, multiple scale wavelet decomposition may be used. In such a case, the access of wavelet coefficients at state x0 is achieved either through another PIWS engine at a coarser resolution level, or through decoding the wavelet coefficients from the compressed bitstream. The flowchart of a 3 scale PIWS engine with pyramidal wavelet decomposition can be shown in Figure 7. At first, slits are accessed by the rendering engine at level 1. Through the recursive access function, wavelet coefficients covering the slits are accessed. For the low pass subband in the horizontal, vertical and slit directions, a second level PIWS engine is used for coefficient access. The accessed coefficients in the other subbands are inverse quantized and entropy decoded. The second level PIWS engine may call a third level PIWS engine for the access of the low pass band, and the inverse quantizer and entropy decoder for the access of the rest subbands. PIWS ensures that a minimum amount of computation is performed for the access of a set of slits.

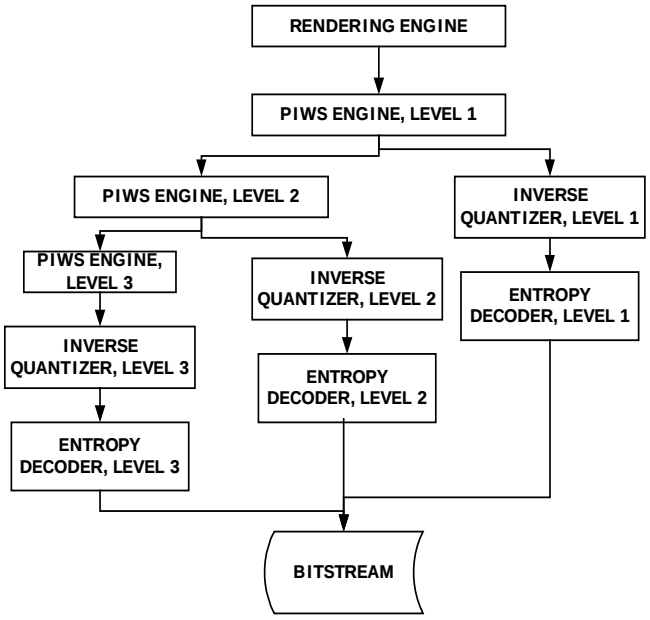

Figure 7 Multiscale progressive inverse wavelet synthesis.

4. EXPERIMENTAL RESULTS

Extensive experiments have been performed to demonstrate the effectiveness of the progressive inverse wavelet synthesis (PIWS) algorithm. The testing platform is a Pentium III PC running at 500MHz. All timing tests are collected on the concentric mosaic scene *Kids* (Figure 8), which is comprised of 1462 shot images with resolution 352x288, totaling 424MB. The wavelet decomposition structure in use is the 4-level mallat decomposition, where wavelet transform is applied along all three directions, and only the low pass subband of all three directions is further decomposed in the next level. However, other wavelet decomposition structures that yield better compression performance may be used as well. An optimal wavelet packet structure suggested by [2] uses 4 level decomposition along the *y*-axis (vertical) followed by 4 level mallat structure in each resulting *(x, z)* layer. It leads to even simpler design than the current full mallat decomposition, because we may simply apply the vertical inverse lifting as an end step and treat the decompression problem as two-dimensional.

Figure 8: Concentric mosaic scene *Kids*.

In the first experiment, we compare the proposed PIWS decoder with two benchmark algorithms. We assume that all wavelet coefficients are already decoded, so that only the computation complexity of the inverse wavelet transform is investigated. The first benchmark algorithm is a simple slit decoder (SSD), where each slit accessed by the rendering engine is independently inverse wavelet transformed with no cache. The second benchmark algorithm is a block selective decoder (BSD). BSD maintains a block cache, and decodes slits block-by-block. The block used in BSD consists of 32x32 slits. SSD and BSD may be used instead of PIWS to provide access to slits requested by the rendering engine, however, as demonstrated below, they are not as efficient as PIWS.

We time the speed to fully decode the concentric mosaics, as well as the speed for actual concentric mosaic viewing operation. The speed is measured in number of slits decoded per second. The larger the number is, the faster the decoder. The speed when an entire concentric mosaic scene is decoded is listed in the first row of Table 4. We observe that the decoding speed of PIWS is slightly faster than BSD (22% speed up), and is much superior to SSD (49 times faster). Although PIWS is designed for the access of random slits, it is also good at the accessing of chunk of data, and does not lose to BSD. The accessing and decoding of individual slit (SSD) is not a good option, as the computation speed is much slower than both PIWS and BSD.

We next investigate the decoding speed in an actual concentric mosaic wondering scenario. Three motion passes of the viewer are simulated, i.e., rotation, forward, and sidestep modes, as shown in Figure 9. In the rotation (RT) mode, the viewpoint is at the center of the circle and rotates *0.006* radians per view. In the forward (FW) mode, the viewpoint starts at the edge of the inner circle and moves forward along the optical axis of the camera. In the sidestep (ST) mode, the viewpoint moves sidestep perpendicular to the optical axis of the camera. The accessed slits associated with the three modes are drawn on a 2D slit plane and shown in Figure 10, where the horizontal and the vertical axes are the angular and radius indices of the concentric mosaics, respectively. The slit plane is exactly the data plane shown in Figure 6. We draw the accessed slits for two views of the motion modes. The RT mode accesses a set of slits parallel to the angular axis. As the viewpoint rotates, the trace slowly shifts to the right. The FW mode accesses a set of slits along a line segment. As the viewpoint moves forward, the line segment rotates along the slit plane. The ST mode accesses a set of slits along a bending curve, and as the

viewpoint moves sidestep, the curve gradually flattens towards the radius axis. The ST mode is the most computation consuming motion, and the RT mode is the cheapest.

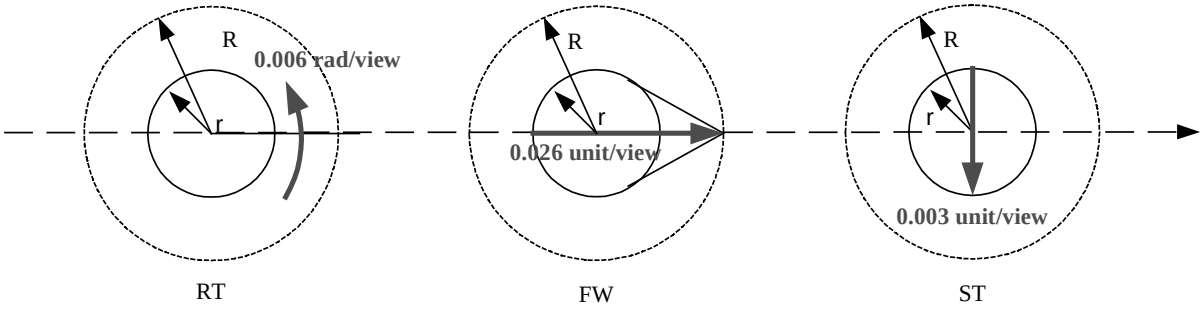

Figure 9 Three kinds of movement in concentric mosaics

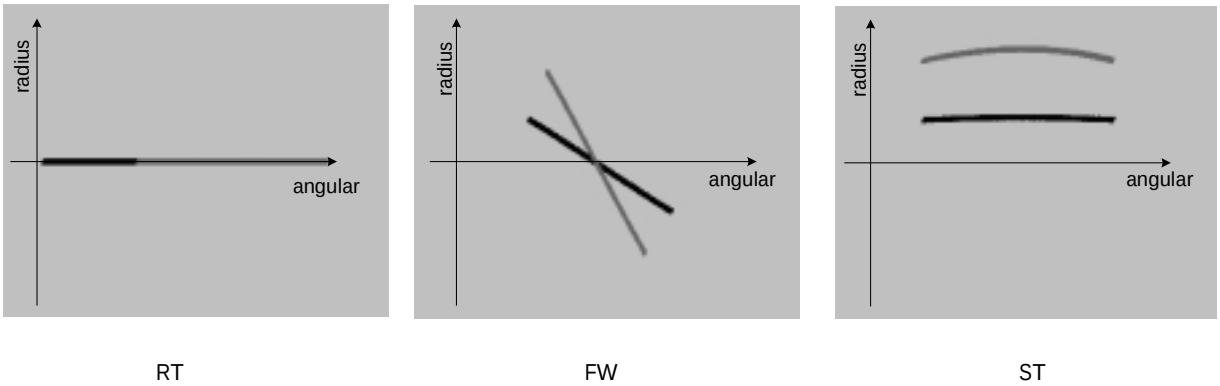

Figure 10 Access slits associated with three motion modes in the concentric mosaics. The slits of two views are drawn.

Rows 2-4 of Table 4 give the average number of slits rendered per second for motion mode RT, FW and ST, respectively. It is observed that PIWS outperforms SSD by 35 to 44 times, and outperforms BSD by 2 to 5 times. The SSD rendering time has been relatively constant since no matter what access pattern is, individual slit is decompressed and then rendered. It is the slowest as a huge number of inverse wavelet lifting is performed to decode the slits. Although BSD is pretty efficient to decompress the entire concentric mosaic data set, it is not suitable for JIT rendering, as the decoding and rendering speed of BSD is dramatically lower than that of PIWS. The decoding speed of BSD slows down by a factor of 5.6 times in the RT mode, and 1.6 times in the ST mode. This is because in the BSD algorithm, additional decoding operations are needed for non-accessed coefficients in each block.

Table 4 Comparison of decoding speed

	PIWS (Slits/Sec)	SSD (Slits/Sec)	BSD (Slits/Sec)
Full Decompression	9756	197	8000
RT: 500 views, 4196 slits	7160	204	1424
FW: 250 views, 8630 slits	7583	202	3512
ST: 50 views, 12403 slits	9260	209	5044

For the comparison algorithms, the time used to render each individual view of the ST mode is further plotted in Figure 11. The horizontal axis is the rendered frame, and the vertical axis is the decoding time for a specific view. It is observed that the SSD is very slow, as it takes on average 1.2 second to render each view. The rendering time of BSD fluctuates greatly from 0ms to 200 ms, as sometimes, there is a cache miss, and large number of blocks is recovered, and some times, all the accessed slits are available and a view can be rendered in no time. The viewing experience of BSD is thus not smooth. PIWS provides a very stable solution as except the first view, on average a view is rendered every 27.1ms. A smooth viewing experience is thus provided. The results clearly demonstrate the superiority of PIWS as it achieves a minimum number of computations required to recover the current view.

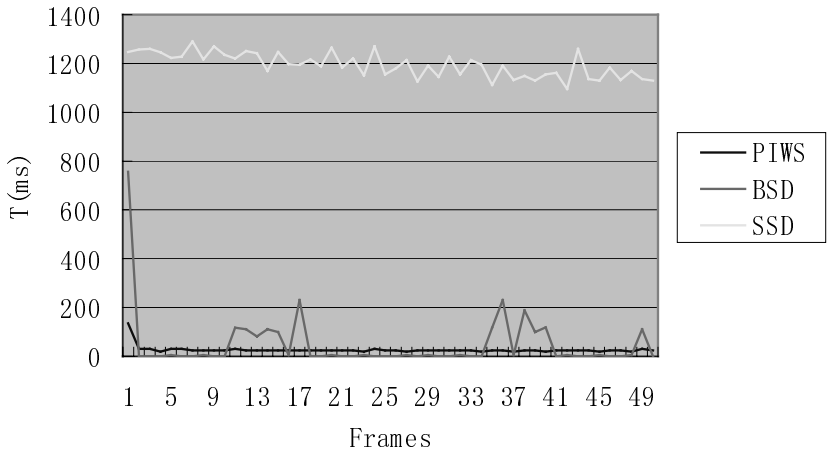

Figure 11 Timing curves for the translation motion of concentric mosaics

In the second experiment, we compare the rendering speed of PIWS with two existing compression and rendering systems for the concentric mosaics: the spatial vector quantization (SVQ) scheme [1] and the reference block coding scheme (RBC)[11]. Both the point sampling (PS) and bilinear interpolation (BI) rendering modes are tested. We observe from Table 5 that the rendering speed of PIWS is slower than SVQ for 26.8% and slower than RBC for 11.5%, especially in the mode ST, where more new slits are accessed. However, the rendering speed is acceptable for a 3D wavelet algorithm.

Table 5 Overall rendering speed measured in frames per sec: VQ, RBC vs. PIWS

Mode/Algorithm \ Rendering setting		PS	BI
RT	VQ	19.7	16.3
	RBC	16.8	13.9
	PIWS	17.6	14.3
FB	VQ	19.0	15.8
	RBC	16.4	13.9
	PIWS	15.8	13.5
ST	VQ	17.7	14.9
	RBC	12.5	10.9
	PIWS	7.9	7.3

6. CONCLUSION

A progressive inverse wavelet synthesis (PIWS) algorithm has been proposed to achieve a minimum computation for the random access of 3D wavelet compressed data. With PIWS, views of the concentric mosaics can be rendered reasonably fast. Although the PIWS is designed for the rendering of the concentric mosaics, with 3D wavelet transform and biorthogonal 9-7 lifting wavelet, it can be extended to other applications with data access to the compressed wavelet, to other wavelet dimension and other lifting filters.

ACKNOWLEDGEMENT

The authors would like to thank Mr. Lie Gu for his generous help in collecting the experimental results.

REFERENCES

[1] H.-Y. Shum and L.-W. He. "Rendering with concentric mosaics", *Computer Graphics Proceedings, Annual Conference series (SIGGRAPH'99)*, pp. 299-306, Los Angeles, Aug. 1999.

[2] L. Luo, Y. Wu, J. Li, and Y.-Q. Zhang, "Compression of concentric mosaic scenery with alignment and 3D wavelet transform", *SPIE Image and Video Communications and Processing*, vol. 3974, SPIE 3974-10, San Jose, CA, Jan. 2000.

[3] I. Ihm and S. Park, "Wavelet-based 3D compression scheme for very large volume data", *Graphics Interface*, pp. 107-116, Jun. 1998.

[4] I. Ihm, and S. Park, "Wavelet-based 3D compression scheme for interactive visualization of very large volume data", *Computer Graphics Forum*, Vol.18, No.1, pp.3-15, Mar. 1999.

[5] C. Bajaj, I. Ihm, and S. Park, "Making 3D texture practical", *Pacific Graphics'99*, pp.259-268, Seoul, Korea, Oct. 1999.

[6] F. F. Rodler, "Wavelet based 3D compression with fast random access for very large volume data", the *Seventh Pacific Conference on Computer Graphics and Applications*, Seoul, Korea, Oct. 1999.

[7] E. Ordentlich, D. Taubman, M. Weinberger, G. Seroussi, M. W. Marcellin, "Memory efficient scalable line-based image coding", *IEEE Data Compression Conference*. Snowbird, Utah, March 1999.

[8] P. Cosman and K. Zeger, "Memory constrained wavelet-based image coding", *Signal Processing Letters*, vol. 5, pp 221-223, Sept. 1998.

[9] W. Sweldens, "The lifting scheme: A new philosophy in biorthogonal wavelet constructions," in *Wavelet Applications in Signal and Image Processing III*, pp. 68-79, Proc. SPIE 2569, 1995.

[10] Verification model ad-hoc, "JPEG 2000 verification model 5.0", *ISO/IEC JTC1/SC29/WG1/N1420*, Oct. 1999.

[11] C. Zhang, J. Li, "Compression and rendering of concentric mosaics with reference block codec(RBC)", *SPIE Visual Communications and Image Processing 2000*, SPIE 4067-05, Perth, Australia, Jun. 2000.

[12] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps", *J. Fourier Anal. Appl.*, vol. 4, pp. 247-269, 1998.