A Hybrid Adaptive Normal Map Texture Compression Algorithm

Bailin Yang
State Key Lab of CAD&CG,
Zhejiang University, 310027 Hangzhou, China
Zhejiang Gongshang University,
Hangzhou, 310035, China
ybl@cad.zju.edu.cn

Zhigen Pan
State Key Lab of CAD&CG,
Zhejiang University 310027 Hangzhou, China
zgpan@cad.zju.edu.cn

Abstract

Normal mapping can reveal the details of complex geometry object and improve the photorealistic when rendering. However, GPU should compress a great number of normal textures to save the valuable video memory and limited bandwidth just as the common texture compression technology. This paper proposes a Hybrid Adaptive Normal Map Texture Compression Algorithm, which utilizes three useful tactics, which are TSSA, BlockIndexTable and I3DC, to compress the normal map texture according to its different features such as a great many of, few or fewer zero texels existed in the normal map texture. The experimental results and comparisons show that it can reach the compression rate of 5: 1 or more and achieve better image result than others'. What's more, it will be a good candidate for hardware implementation because of its low cost.

1. Introduction

Normal Mapping [9], extended to Bump Mapping [2], is as a key technology to enhance the roughness and wrinkles of the surface with small amount of polygon data [6]. With the wide use of normal mapping, it is supported by DirectX, OpenGL and OpenGL ES by default now. What's more, hardware accelerated normal mapping chips also have been developed.

However, the following aspects should be mentioned with the introduced of Normal Mapping. First, the burden of GPU' process ability will risen. Second, the requirement for data storage space is also high. Finally, the bottleneck of the video card's bandwidth will be aggravated by the transmission of a great deal of texture data.

Fortunately, texture compression technology such as the DXTC, FXT and IPACKMAN [11] etc al. can solve the above problems up to certain degree. Furthermore, these common texture compression technologies also be adopted or modified to compress the normal map texture such as the Swizzled DXT5 [7] and 3DC [1].

This paper focuses on how to compress the normal map texture effectively for GPU hardware implementation. We propose the Hybrid Adaptive Normal Map Texture Compression Algorithm to compress the normal map by hardware implementation. This paper is organized as the follows. In section 2, related work is discussed. The HANMTC algorithm will described in detail in section 3. After that, experimental results are illustrated and discussed indicating the improvements using our new approach. Finally, Section 5 concludes with the future work.

2. Related Work

Texture compression, which is an effective method, can save valuable GPU memory and limited bandwidth. Delp and Mitchel [4] proposes the Block Truncation Coding, or BTC, which is a common technique for hardware implementation of texture compression. As an extension to the BTC, Campbell [3] designs the CCC . Iourcha [8] et al. design S3TC, which is called DXTC by Microsoft. Fenny [5] proposes a novel technique, which provides high quantity texture image. However, all of these techniques and algorithms are not good for normal map texture because they are designed for common texture not for normal maps.

Akenine-Moller and Strom [11] proposes IPACKMAN, which is designed for mobile device' GPU and can be employed to compress normal texture. Unfortunately, it can not achieve good result for high frequency of normal maps. Palletize also can achieve high compress ratio than DXT but it require high storage space. Reference [10] utilizes JPEG technique to compress normal maps, but it still need high hardware cost.

Swizzled DXT5, which improves DXT5, is designed target for normal maps. 3DC [1] developed by ATI Company provides 3:1 compression rate and get better image

Proceedings of the 16th International Conference on
Artificial Reality and Telexistence--Workshops (ICAT'06)
0-7695-2754-X/06 $20.00 © 2006 IEEE

result. Experimental result shows that 3DC is better than Swizzled DXT5. Nevertheless, 3DC can not represent high precision normal maps resulting in the errors between the decompressed image and source image. Furthermore, the compress ratio is not high so as to the compressed texture data still occupying a great deal of storage and limited bandwidth.

3. Hybrid Adaptive Normal Texture Compress Algorithm

In this section, our new hybrid adaptive normal texture compression system will be presented. First, we describe the underlying design and discuss the motivation for the design choices. Then follows subsections for the detail implementation of there different compression tactics.

3.1. Basic Motivation and Design

There are two main compression techniques including lossless and lossy compression. Run Length Code is a kind of lossless compressing coding technique which utilizes the basic facts that there are a large number of continuous same color pixels in some certain images and will achieve a high compress rate. However, this method is not suitable for hardware implementation since it will bring more hardware cost.

As we all know, Block-Based texture compression technique, as one kind of lossy compression technique, is the general method for hardware implementation of texture compression. Block-Based texture compression method compresses and decompresses all origin texture data instead of taking the feature of a large number of same texels in some textures into account. The advantage of this method is that it can decompress the texture RealTime. On the contrary, its shortcoming is low compression rate and low quality.

As the first section said, normal texture is widely used in 3D game and improves the realism of the game. The value of X, Y and Z is represented by R, G and B sector of the color. The mapping is denoted by Equation 1.

$$(x, y, z) = \frac{2}{255}(R, G, B) - 1 \qquad (1)$$

The range of both the R and G's value is from 0 to 255 and the corresponding value of X and Y is from -1 to 1. Since the direction of normal always points to the outside of objects, the value of Z is always positive. As a result, the range of the Z's value is from 127 to 255 and the corresponding value of Z is from 0 to 1. We can discern that the color of the normal map texture is blue and purple.

For clarity, we define the following:

ZeroTexel : The texel's RGB value is (127,127,255).

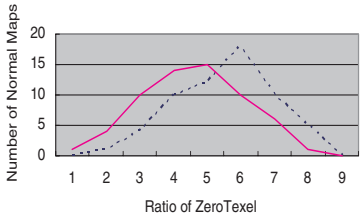

Figure 1. The number of different ratios of both the *ZeroTexel* and *ZeroBlock*.The dot line is the *ZeroTexel* and real line represents the *ZeroBlock*.

ZeroBlock$(m \times m)$: The block has $m \times m$ texels and all of them are ***ZeroTexel*** .

ZeroTile$(n \times n)$: The tile has $n \times n$ blocks and all of them are ***ZeroBlock***.

Normal mapping mainly represents the details of the model. As a fact, the ratio of the details for most 3D model is not high. We take 60 normal textures of Doom3 game as our test suit. Experimental result shows that there exist many ***ZeroTexels*** in lots of normal map texture. If we can utilize this feature fully and employ the theory of the RLF, which does not store the ***ZeroTexel***s instead of only marking them, the compression rate will be increased.

From the Fig. 1, we can see that the number of different ratios of both the ***ZeroTexel*** and ***ZeroBlock*** among the whole samples. In this experiment, the ***ZeroBlock*** contains 16(44) texels. It shows that the textures, whose ***ZeroTexels*** ratio is from 50% to 70% percents, make up the majority of the test samples. If we do not store these ***ZeroTexels***, the compression rate will be increased by twice or more.

The RLF is not a good candidate for hardware texture compression so that we still make use of Block-based compression technique to compress the normal map texture. From the trend of real line in Fig. 1, the number of texture, whose ***ZeroBlock*** ratio is 40%-50%, is about 15 to 20. Therefore, we can design a high compression rate texture compression algorithm called Hybrid Adaptive Normal Texture Compress Algorithm, which is suitable for hardware implementation while utilize the advantages of RLF and Block-Based technique.

The above algorithm is designed for the basic fact that there are over half of the number of total texels ***ZeroTexels*** in certain normal texture. Whereas, there still exist some normal textures that have few or not many ***ZeroTexels***. In order to make our compression algorithm be applied for all normal texture feasibly, we also provide two different tactics to compress these kinds of texture.

Proceedings of the 16th International Conference on
Artificial Reality and Telexistence--Workshops (ICAT'06)
0-7695-2754-X/06 $20.00 © 2006 IEEE

If the texture has only fewer ***ZeroTexels***, we improve the 3DC algorithm called I3DC, which can increase the compression quality, to compress this kind of normal texture. For the texture having not many ***ZeroTexels***, we utilize the BlockIndexTable algorithm to compress the normal texture. The BlockIndexTable algorithm introduces a Block Index Table to record the flags 0 or 1, which means whether this block is ***ZeroBlock*** or not. We will discuss the details of the two algorithms in section 3.2 and 3.3.

In conclusion, we develop a Hybrid Adaptive Normal Texture Compress Algorithm which can cope with different normal map. While the normal maps have lots of ***ZeroBlock***, we choose the Tile-based Spare Store Compression Algorithm to compress them. Otherwise we will select one of I3DC and BlockIndexTable algorithm to compress the normal map. If there exists few ***ZeroBlock*** we employ the BlockIndexTable, otherwise we select the I3DC.

3.2. I3DC Algorithm

3DC algorithm represents each index of texel with 3 Bits and the range of two interpolations' difference is [0,16]. Then, the upper and lower bound of the compression error is 0 and 7 respectively. Therefore, the shortcoming for 3DC is it can not represent high precision normal map texture because it will bring compression errors.

In this paper, we propose an improved 3DC version, which will decrease the compression error. We adopt 4 bits not 3 bits for 3DC to denote the index of every texel. First, we get the max and min value in the 4×4 texels block and gain the rest 14 interpolation values between the max and min. After that, we compute the indexes for these 16 texels. In a result, the range of compression error is decreased from 0 to 3 not from 0 to 7 for 3DC. Furthermore, we only store the Delta values requiring 4 bits instead of the Min values requiring 8 bits for X and Y thus will save one byte for every block when storing. Another advantage for only storing Delta for X and Y is that it will reduce the decompression hardware cost because there are only few add arithmetic operations executed for obtaining the value of each texel.

The bit layout of a 4×4 block for 3DC (top) and I3DC(down) is illustrated in Figure 2, and each block contains the following information:

X0,Y0X15,Y15 : Index for each texel in current block;

Xmax,Ymax : Maximum X and Y value in current block;

Xmin, Ymin : Minimum X and Y value in current block;

DeltaX,DeltaY: Difference between two interpolations.

It shows from Fig. 2 that I3DC require more bits than 3DC. However it can represent high precision normal map texture and this disadvantage will be made up by the other aspects.

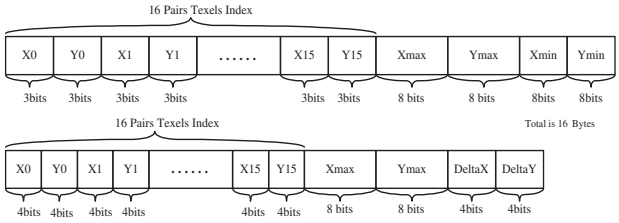

Figure 2. The bit layout of a 44 block for 3DC (top) and I3DC(down).

3.3. BlockIndexTable Algorithm

When the number of ***ZeroTexel*** is neither large nor fewer, we introduce the BlockIndexTable Algorithm to compress the normal texture. This tactics also makes use of the feature that there exist few ***ZeroBlock*** in normal texture. For this method, we pre-compute the whole normal map texture and build a block index table in compression stage. In this block index table, we record and set a flag for each block, which is 0 if this block is ***ZeroBlock*** and 1 if not ***ZeroBlock***. In addition, we also store each none ***ZeroBlock*** in a continuous memory block as I3DC does.

Fig. 3 illustrates a hardware diagram for BlockIndexTable Algorithm. Below we will describe in detail how a textel is decompressed.

- First, the x,y position of current texel need to be obtained. We compute the block index for this texel and get the ***ZeroBlock*** flag in the BlockIndexTable.
- The next step is to get the texel value. If the ***ZeroBlock*** flag is 0, we obtain this texel's value (127,127,255) directly. Otherwise, we should get its color value from the I3DC block.
- To get the texel's color from certain block, we must first decide which block the current texel locates in among the continuous memory. For simplicity, we also store the zero blocks in our memory because locate the certain position of this block in the whole texture memory will increase the hardware cost. It means that we may obtain the block's head address as I3DC does.
- Finally, we will compute the texel's color from this I3DC block as the decompression of I3DC.

3.4. Tile-based Spare Store Compression Algorithm (TSSA)

Reference [10] lists four factors that should be considered when evaluating a texture compression scheme: Decoding speed, Random Access, Compression rate, visual

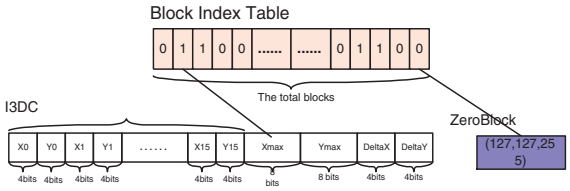

Figure 3. The bits layout of BlockIndexTable.

quality and Encoding Speed. Among of these four factors, the Decoding speed and Random Access are very important for hardware implementation.

In order to meet the above requirements, we propose the Tile-based Spare Store Compression Algorithm (TSSA). In this algorithm, we do not store the ***ZeroBlock*** for saving the valuable video memory and just store the details of 3D model with high precision for improving the image quality. Furthermore, we adopt both the Tile technique and block-based encoding method to meet the needs of fast decoding speed and random access.

One problem, however, is occurred when we decompress the texel while use the TSSA. We can not decide whether the texel or block is Zero or not. One solution to this problem is that we build an index table which is consisting of Zero flag and the address of this block. Thus, the index table will occupy a large number of video memories. For instance, the size of the index table is $4K \times 4Bytes + 4K/8 = 16.5K$Bytes if the texture is$256 \times 256$ and the block size is 4×4. For solving the above problems, we introduce the tile mechanism which is widely employed in rasterization and texture mapping [12] so as to increase the efficiency and speed. In our TSSA, we divide the Texture space into mn Tiles and each tile also been divided into nn blocks in order to decrease the hardware computation cost and achieve fast decoding speed and Random Access.

Figure 4 illustrates the diagram of compression and decompression stage. We will depict the compression and decompression stage in detail in the following.

Compression stage

- First, we divide the normal texture into n tiles and each will contain the same number of blocks. The TilesFlag is an integer array and we calculate the number of None ***ZeroBlock*** of each tile and store it into the next element of the TileFlag array. For example, we store 0, 50 ,130 in Tile 0 ,Tile 1 and Tile 2 to represent the head address of this tile stored in the TileCompressedData.

- After that, we build a BlockIndex table for every Tile. In this BlockIndex table, we employ the same method as the BlockIndexTable algorithm. The difference is that TSSA algorithm builds the BlockIndex only for certain Tile not whole texture.

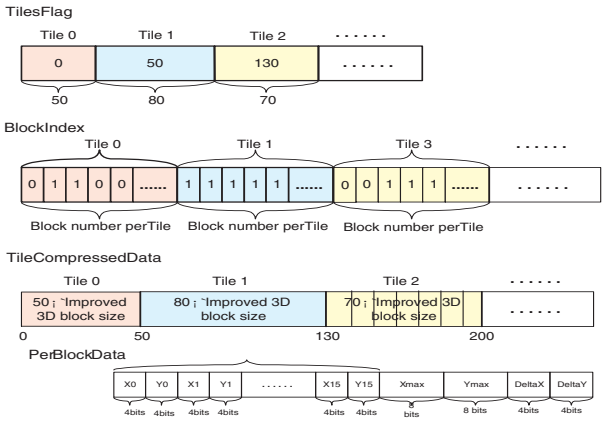

Figure 4. This diagram shows the decompressor of TSSA .

- Finally, we will store the actual texel compressed data in the TileCompressedData, which constitutes of all of the non ***ZeroBlock*** for all tiles. As the BlockIndexTable algorithm, every block is stored as I3DC format.

Decompression stage

- While the current texel's X and Y position is obtained, we can access the TileIndex and BlockIndex directly because we employ the block-based compression scheme and each tile's size is also same.

- The Next step is to acquire the block flag value 0 or 1 from the BlockIndex. If it is 0, which means the texel is a ***ZeroTexel***, we will return the texel's color value (127,127,255). Otherwise, we continue to execute the third step.

- The third step is to get the actual memory position of the current texel's block in the TileCompressedData memory. How to get the position? The position is calculated by the sum of the number of None ***ZeroBlock*** before the current BlockIndex and TileIndex. We have got the tile index in the first step. Obtaining the number of None ***ZeroBlock*** is by counting the number of the 1 flag before current block index in this blockindex flag of current tile. For example, if the block index is 11001010 and the current block index is 4, we can get the None***ZeroBlock*** 2 through employing serials of logical operations for 4 and 11001010. Because every block is fix size, we can obtain the first address of block current texel belongs to easily in the TileCompressedData.

Proceedings of the 16th International Conference on
Artificial Reality and Telexistence--Workshops (ICAT'06)
0-7695-2754-X/06 $20.00 © 2006 IEEE

- In the end, we decompress the block as the I3DC does and get the final color of this texel.

4 Experimental Results and Comparisons

In this section, we present results showing the compression rate, image quality and decompress hardware cost comparing with 3DC algorithm. Most of the normal textures are taken from the DOOM 3 game. For convenience, we select the images whose sizes are $256 \times 256 \times 24$.

4.1 Compression rate

In this subsection, we analyze the compression rate of TSSA algorithm. From Figure 4, we can see that the compressed data mainly consists of three parts, which are *TilesFlag(TilesF)*, *BlockFlag(BlockF)* and *TileCompressedData(TileCD)*. The compression rate for the compressed image comparing to source image is demonstrated in Eq 2.

$$TSSACompressed = (TilesF + BlockF + TileCD)/SI \quad (2)$$

The memory space *TilesFlag* and *BlockFlag* consumed can be ignored comparing to *TileCompressedData*. Thus, we can simplify the Eq 2.to Eq 3. In Eq. 3, the number 19 represents the memory size of each block and *ZeroBlockRatio* denotes the ***ZeroBlock*** ratio of current test image.

$$TSSACompressed = 19 \times ZeroBlockRatio(16 \times 3) \quad (3)$$

From our experiments for 24 test samples, it can be concluded that the TSSA has lower compression rate value 0.1985437(about 5:1) than 3DC's 0.3333(about 3:1).

4.2. Compression Result

We implement the 3DC, S3TC and HANTC algorithm respectively with soft simulation. Fig. 5 demonstrates the different results of one normal texture image from the test suit for these algorithms. The results show that our HANTC algorithm can achieve the best effects.

4.3. Decompression Cost

For our HANTC algorithm, we analyze the decompression cost for I3DC, BlockIndexTable and TSSA respectively. For I3DC, the decompress cost is smaller than 3DC because we only use the addition arithmetic operations to achieve the texel's color. Evidently, the decompression cost for BlockIndexTable is lower than 3DC. For example, the decompression cost is reduced to 80% when there are about 20% percents of ***ZeroBlocks*** in the image.

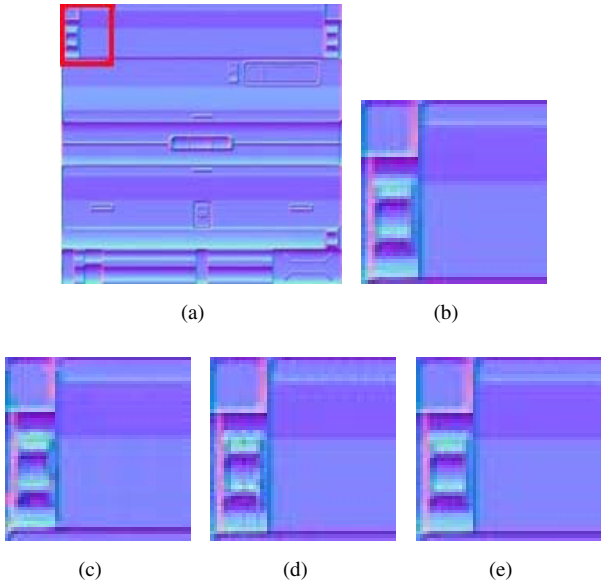

Figure 5. a) Source Texture. b) the magnification of origin. c) S3TC d) 3DC e) HANTC

However, the decompression cost for the TSSA is complicate. The total decompression cost for the TSSA consists of the following:

TileCost(TileC): Cost for obtaining the tile number;

BlockFlagCost(BlockFC): Cost for obtaining the block flag in tile;

BlockDecompressCost(BlockDC): Cost for obtaining the final texel color from the current block.

Eq. 4 describes how to get the total decompression cost for the TSSA. In Eq. 4, the *ZeroBlock* Ratio depicts the ratio of ***ZeroBlocks*** in this test image. The cost for *TileCost(TileC)* and *BlockFlagCost* is smaller than *BlockDecompressCost* because there are only additions and bits operations in *TileCost* and *BlockFlagCost*. What's more, *BlockDecompressCost* always takes up the majority of the total cost. When the *ZeroBlockRatio(ZBRatio)* is 0.5423%, the decompression cost for TSSA is 94% comparing to 3DC.

$$TotalCost = TileC + BlockFC + BlockDC \times ZBRatio \quad (4)$$

5. Conclusion and Future Work

This paper proposes a Hybrid Adaptive Normal Map Texture Compression Algorithm, which utilizing the specialty that there exist many ***ZeroTexels*** in some certain normal texture. In order to cope with all of the normal texture, we utilizes three useful tactics to compress the normal texture image according to the different characteristic such as a

Proceedings of the 16th International Conference on
Artificial Reality and Telexistence--Workshops (ICAT'06)
0-7695-2754-X/06 $20.00 © 2006 IEEE

great deal of, few, or fewer zero texels existed in the normal mapping texture. The experiment results show that this algorithm can achieve high compression rate and better image result, access the texel randomly and decompress the block RealTime.

However, there are several possible improvements to the HANTC scheme. We can adopt the post-process to decrease the errors between the blocks to enhance the image quality. Furthermore, we can employ the frequency analysis method as fenny [5] does to improve our algorithm.

Acknowledgment

This project is co-supported by National Natural Science Foundation of China (Grant No 60533080).

References

[1] *ATI RADEON X800 3Dc white paper*. http://www.ati.com/products/radeonx800/3DcWhitePaper.pdf, 2004.

[2] J. F. Blinn. Simulation of wrinkled surfaces. volume 12, pages 286–292. the 5th annual conference on Computer graphics and interactive techniques, 1978.

[3] G. campell. Two bit pixel full color encoding. volume 22, page 215C223. SIGGRAPH, 1986.

[4] E. Delp and O. Mitchell. Image compression using block truncation coding. *IEEE Transactions on Communications*, 9(9):1335C1342, IEEE Transactions on Communications.

[5] S. Fenny. Texture compression using low- frequency signal modulation. volume 22, page 84C91. Graphics Hardware, ACM Press, 2003.

[6] J. D. Foley and A. Dam. *Computer Graphics PRINCIPLES AND PRACTICE, 2nd Edition*. Addison-Wesley Publishing Company, 1996.

[7] S. Green. *Bump Map Compression*. Nvidia Corporation, 2004.

[8] K. Iourcha. *System and Method for Fixed-Rate Block-based Image Compression with Inferred Pixels Values*. In US Patent 5,956,431, 1999.

[9] M. J. Kilgard. A practical and robust bump-mapping technique for todays gpus. Game Developers Conference, Advanced OpenGL Game Development, 2000.

[10] R. Nakamura, T. Yamasaki, and K. Aizawa. Jpeg optimization for efficient bump map compression. volume 22, pages 11–17. Proc. of the 2005 IEICE General Conf, ACM Press, March 2005.

[11] J. Storm and T. A. Miller. Graphics for the masses a hardware rasterization architecture for mobile phones. volume 22 of *3*, page 801C808. ACM Transactions on Graphics, 2003.

[12] L. Y. Wei. Tile-based texture mapping on graphics hardware. volume 22, page 84C91. France, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, August 29-30 2004.