Adaptive Normal Map Compression for 3D Video Games

Alexander Wong
a28wong@engmail.uwaterloo.ca

William Bishop
wdbishop@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1.

ABSTRACT

Next-generation 3D video games make use of normal maps to improve the realism and the visual detail of scenes and models. The use of normal maps typically leads to increases in data storage and bus bandwidth requirements. To cope with the increased demands of normal maps, compression techniques can be used. Unfortunately, most texture compression techniques are poorly suited for normal map compression. The application of such techniques on normal maps often leads to a degradation of visual quality. This paper presents an adaptive approach to the task of compressing normal maps used by next-generation 3D video games. The proposed method attempts to retain visual detail while reducing storage requirements. Experimental results demonstrate overall compression performance and visual quality that is superior to the performance of existing techniques. The proposed method can be used to deliver improved photo-realism in 3D video games.

Keywords: normal maps, adaptive compression, 3D video games

1. INTRODUCTION

One of the hottest graphics techniques being adopted by next-generation 3D video games is the application of normal maps. This technique can be used to significantly improve the visual quality of a graphical object by making its surface appear much more detailed than a flat surface. Figure 1 illustrates the application of a normal map on a sphere. When a normal map is applied to the sphere, the surface of the object appears to have significantly more geometric detail than the original object, despite the fact that the underlying model is identical. Furthermore, the use of normal maps is often a more efficient approach for improving the detail of a model than increasing the number of polygons used. This makes the technique suitable for real-time use in 3D video games. Furthermore, normal maps are used in state-of-the-art 3D techniques such as parallax occlusion mapping [1] and steep parallax mapping [2], which can be used in future video games to bring real-time 3D game graphics to the next level.

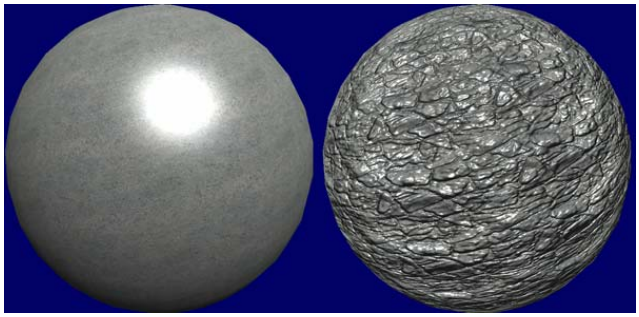

Figure 1: Left: Sphere with texture mapping
Right: Sphere with texture mapping and normal map bump-mapping

One of the major drawbacks to using normal maps is that it significantly increases memory and bandwidth requirements. This is particularly problematic in the case where a normal map is generated from a high-polygon model to cover a low-polygon model in its entirety. Therefore, a way to compress normal maps is highly desired. While different techniques have been proposed for compressing images to reduce bandwidth and storage requirements, most are not well-suited for handling normal maps in a real-time 3D video game. For example, transform-based techniques such as JPEG [3] and JPEG2000 [4] do not permit random access to individual texels in a texture map. Furthermore, these techniques are relatively complex to implement and computationally expensive.

A number of different texture compression techniques have been used in 3D video games. DXTC

(DirectX Texture Compression) is a family of texture compression formats based on the Color Cell Compression (CCC) method [5]. In particular, the DXT-1 format can be used to compress a RGBA texture at a ratio of 6:1 and the DXT-5 format can be used to compress a RGBA texture at a ratio of 4:1. The main advantage of DXTC is that it is part of the Direct3D standard so it is well supported by modern 3D graphics devices. While DXTC is effective at compressing color textures, it performs poorly on normal maps since important details are often lost. The compression artifacts are further amplified when normal maps are used in conjunction with shaders that include specular reflections [6].

Recently, ATI Technologies made an attempt to address this issue by introducing the 3Dc compression format [7]. Based on BTC (Block Truncation Coding) [8], the 3Dc compression format is able to achieve better visual detail than DXTC when compressing normal maps. However, it is only able to achieve a compression ratio of 4:1. The reason that a higher compression ratio is not achieved is that the same amount of data is used to represent smooth regions and detailed regions in a normal map. In reality, smooth regions can be stored in less space than detailed regions without a noticeable loss in visual detail.

The main contribution of this paper is an efficient adaptive compression algorithm for compressing normal maps in 3D video games. This method reduces the amount of information needed to represent normal maps while achieving a high level of visual detail. This allows for future 3D video games to deliver a higher level of visual detail and photo-realism. In this paper, the underlying theory behind adaptive normal map compression is presented and explained in Section 2. An implementation of the proposed algorithm is presented in Section 3. The testing methods and test data are outlined in Section 4. Finally, experimental results comparing the proposed algorithm with DXTC and 3Dc are discussed in Section 5, and conclusions are drawn in Section 6.

2. THEORY

Before outlining an implementation of the proposed normal map compression algorithm, it is important to discuss the theory behind the key components of the algorithm. First, an overview of normal maps is presented. Second, the underlying theory behind the BTC compression scheme is explained. Finally, the concept of adaptive normal map compression is introduced and described to justify its use in reducing storage requirements while retaining visual detail.

2.1. Normal Maps

In normal maps, the normal vector of a surface at a particular location is stored in the form of a raster image. A normal vector can be represented by its *x*, *y*, and *z* components. Thus, surface normal information is typically stored as three separate component channels in an image. Using this information, the direction of light reflections can be computed at a particular point for all light sources. This allows the lighting of a surface to be performed in a much more detailed manner. This gives the illusion that the surface has greater geometric detail than it really does. One popular use of normal maps is to capture surface normal information from a high-polygon model and then apply the normal map to a low-polygon model. This gives the low-polygon model the visual detail of a higher-polygon model without the need for as many polygons. This is illustrated in Figure 2, where a color map and a normal map are applied to a plane. Due to the way the lighting is calculated based on the normal information, the surface appears to contain the geometric details of a high-polygon surface. However, the underlying plane has no real geometry.

Figure 2: A color map and a normal map are applied to a plane

One important characteristic of normal maps is that normal vectors should be unit length. Furthermore, the *z*-component of the normal vector can be assumed to be positive, as it should be pointing out of the surface. Therefore, the *z*-component can be calculated using the *x* and *y* components of the normal vector using the following formula:

$$z = \sqrt{1 - \left(x^2 + y^2\right)} \quad (1)$$

The *z*-component can be calculated using pixel shaders with very little computational cost. By calculating the *z*-component of the normal vector as opposed to storing it, the amount of data required to store a normal map is reduced by one third. The data savings can then be used to represent the *x* and *y* components with greater precision.

2.2. Block Truncation Coding (BTC)

Block Truncation Coding (BTC) is a lossy image compression method that can be seen as the basis of many popular texture compression techniques. In BTC, a grayscale image is divided into a set of smaller blocks. Based on local statistics within a block, two grayscale values are chosen to represent the image, V_{LOW} and V_{HIGH}. One way of choosing the two grayscale values is to find the mean grayscale value of the image. The pixels are then grouped together depending on whether it is greater than the mean value. The mean grayscale value is then found for the pixel group whose values are greater than the mean of the image. This value is then used for $V_{HIGH.}$ Similarly, the mean grayscale value is found for the pixel group whose values are less than the mean of the image. This value is then assigned to V_{LOW}. A bit-mask is created by classifying the pixels within the block relative to the chosen grayscale values. For example, if a pixel's grayscale value is closer to V_{LOW} than V_{HIGH}, then the pixel is assigned a value of 0 in the block mask. Otherwise the pixel is assigned a value of 1 in the block mask. The block mask and the two grayscale values can then be stored and used to reconstruct an approximation of the image block.

To demonstrate the effectiveness of BTC for compressing image data, consider the following example using a 256×256 texture map. Assume that each pixel in the original texture map is represented by a 32-bit value. BTC divides the image into blocks of size 4×4 and represents pixel intensities using 8-bit grayscale values. Therefore, for each block, two 8-bit grayscale values and one 16-bit block mask are stored. This results in a total storage requirement of 32-bits for each block. Since the original image uses $16 \times 32 = 512$ bits of data for each block, BTC compression is able to achieve a compression ratio of 16:1.

There are a number of advantages to using BTC for texture compression. First, it is relatively simple to implement and has a low computational cost. Second, it allows for random access to individual texels without the need to decompress the entire texture. This is important for real-time 3D applications such as video games. Given all of these benefits, the proposed adaptive normal map compression algorithm is based on the BTC compression framework.

2.3. Adaptive Normal Map Compression

To compress a normal map using BTC, each component channel that needs to be stored (*x* and *y*) is compressed separately. One problem with using the basic BTC algorithm on normal maps is the fact that not enough bits are used to provide an accurate representation of the normal vectors. This results in disturbing artifacts such as image banding and blocking artifacts. One simple method of reducing the visual detail degradation caused by compressing normal maps is to increase the number of bits used to store the normal vector information. This is the approach taken by ATI Technologies for the 3Dc texture compression algorithm. While the resultant visual quality is much improved over DXTC and standard BTC, the compression ratio is reduced to 4:1.

A more memory-efficient approach to improving visual detail quality is to allocate more bits only to regions on the normal map that benefit from the additional bits. Furthermore, the number of bits allocated is reduced for regions that can be represented with fewer bits without noticeable visual degradation. Using this approach, the amount of data used to store a normal map is the amount necessary to represent the normal map with a high level of visual detail. This is the approach taken by the proposed algorithm.

To perform adaptive normal map compression, it is necessary to select an evaluation metric for determining the level of compression a block in the

normal map can undergo without noticeable visual artifacts. Therefore, it is important to understand some of the common characteristics of normal maps. Normal maps are used to represent the geometry of a surface. Detail degradation is most noticeable to the human vision system in regions that exhibit a large change in geometry such as edges. Therefore, a good way to quantify such changes is to measure the gradient magnitude on the normal map, as given by

$$G = \sqrt{\left(\nabla_x^{\,2} + \nabla_y^{\,2}\right)} \tag{2}$$

where ∇_x is the partial derivative with respect to the x direction and ∇_y is the partial derivative with respect to the y direction. A high gradient magnitude indicates large changes in geometry on the surface. Therefore, the following measure can be used to classify a block based on gradient magnitude:

$$R = \frac{\sum_x \sum_y G(x, y)}{N(G_{\max})} \tag{3}$$

where N is the number of pixels in a block and $G_{\max}$ is the maximum possible value of G. If a block has a high value of R, then it is necessary to allocate more bits to the block to preserve visual detail. If a block has a low value of R, then the number of bits used can be reduced without noticeable loss in visual detail. To reduce the complexity of the proposed algorithm, the blocks are classified into a small number of classes using fixed thresholds. For example, if the value of R for a block is less than threshold t_A, then the block belongs to class A. Each class requires a fixed number of bits for each block. Furthermore, the class information about a block must also be stored. It is important to note that each component channel must be evaluated separately since the characteristics of each component may be different within the same block.

Since blocks of different classes require different amount of storage to preserve visual detail, it is necessary to extend the standard BTC algorithm to allow for different levels of compression. One effective way is to increase the number of values used to represent a particular block depending on the level of compression required. For example, blocks with a low value of R can be well represented with 2 values while blocks with a high value of R would require 8 values to preserve visual detail. To avoid the issue of having to store additional values, the two values (V_{LOW} and V_{HIGH}) can be used to calculate the remaining intermediate values for the block. The values used to represent components within a block are calculated as follows:

$$Value_i = V_{\text{LOW}} + i\frac{\left(V_{\text{HIGH}} - V_{\text{LOW}}\right)}{N-1},\ i = 0,1,..,N-1 \tag{4}$$

where N is the number of values used to represent a block. A block mask is constructed to indicate the value of a particular texel within the block. The block mask and the two values (V_{LOW} and V_{HIGH}) are stored and used to reconstruct the block in the decompression process.

There are a number of advantages to this approach to improving the visual quality of compressed normal maps. First, since the entire block classification process is performed during the production stage of game development, the computational cost of the proposed algorithm for texture decompression during a game is low. Second, the proposed algorithm retains the ability to allow efficient random access to individual texels without full normal map decompression, despite its adaptive nature.

2.4. Implementation

Based on the theory presented, a practical implementation of adaptive normal map compression can be defined. Assume that the original normal map is stored as a 3-channel image (one channel for each component). Each texel in the image is represented by 32 bits of data. For the proposed implementation, the normal map is broken up into 4×4 blocks. Therefore, there are 16 texels within each block. Since the z-component can be calculated using the x and y components, only the x and y components are stored. Each component channel is compressed separately. For each channel, the blocks are classified into one of 3 classes (LOW, MED, and HIGH) in the following manner:

$$Class(Block_i) = \begin{cases} \text{LOW} & \text{if } R < t_{\text{LOW}} \\ \text{MED} & \text{if } t_{\text{LOW}} \le R \le t_{\text{HIGH}} \\ \text{HIGH} & \text{if } R > t_{\text{HIGH}} \end{cases} \quad (5)$$

where t_{LOW} and t_{HIGH} are thresholds used for block classification purposes. For testing purposes, t_{LOW} and t_{HIGH} were set to 0.07 and 0.18 respectively based on the results of subjective quality tests. Since 3 classes were used, 2 bits were needed per block to indicate the assigned class. The class information can be used to quickly calculate the offset to a particular texel.

For blocks classified as LOW, two 8-bit values and one 16-bit block mask are stored to represent each block for the component channel. Including the 2 extra bits used to indicate the assigned class, a total of 34 bits are required. For blocks classified as MED, two 8-bit stored values, two 8-bit calculated values, and one 32-bit block mask are used to represent each block for the component channel. Including the 2 extra bits used to indicate the assigned class, a total of 50 bits are required. Finally, for blocks classified as HIGH, two 8-bit stored values, six 8-bit calculated values, and one 48-bit block mask are used to represent each block for the component channel. Including the 2 extra bits used to indicate the assigned class, a total of 66 bits are required.

3. TESTING METHODS

To demonstrate the effectiveness of the proposed algorithm in real-world situations, the proposed implementation was tested using four normal maps of size 256×256. These test maps are representative of the types of normal maps used in a 3D video game. A description of each normal map is described below.

- **DOOR:** A normal map of a metallic door texture [9].
- **BRICK:** A normal map for a brick wall texture [10].
- **COBBLES:** A normal map for a cobblestone texture [10].
- **WOOD:** A normal map for a wooden floor texture [10].

For testing purposes, the same classification thresholds are used for all test cases. In a real-world scenario, the artist can adjust the thresholds to provide improved compression performance and/or improved visual quality for a specific type of normal map. The normal maps were compressed using baseline DXTC and 3Dc for purposes of comparison.

4. EXPERIMENTAL RESULTS

The compression performance results are shown in Table I. It can be observed that the proposed method showed noticeably better compression performance over the 3Dc method in all test cases. The overall compression ratio is close to that achieved using DXT1 RGBA. The test cases are shown using the three compression methods in Fig. 3 through Fig. 6. The visual quality of the proposed method is noticeably superior to the DXTC method. The fine details are retained much better using the proposed method. Furthermore, the visual quality of the proposed method is comparable to that achieved using the 3Dc method, while still achieving higher compression performance.

TABLE I
COMPRESSION PERFORMANCE

Map	Compression Ratio	Improvement over 3Dc
DOOR	5.57:1	39.25%
BRICK	6.30:1	57.50%
COBBLES	4.96:1	24.00%
WOOD	6.28:1	57.00%
Overall	**5.78:1**	**44.50%**

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced an efficient and adaptive method for compressing normal maps for use in real-time 3D video games. Experimental results show good compression performance to 3Dc and superior visual quality to DXTC. It is believed that this method can be successfully implemented into 3D video games to improve visual detail of game objects and environments. Future work includes developing a hardware implementation of the proposed method.

ACKNOWLEDGEMENTS

The authors would like to thank Epson Canada and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] N. Tatarchuk, "Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows," in Proceedings of the 2006 symposium on Interactive 3D graphics and games, 2006, pp. 63-2006.

[2] Morgan McGuire and Max McGuire, "Steep Parallax Mapping," presented at *I3D 2005*, http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.html.

[3] G. Wallace, "The JPEG Still Picture Compression Standard," *Comm. ACM*, 1991, Vol. 34, No. 4, pp. 30-34.

[4] M. Boliek, C. Christopoulos, and E. Majani, "JPEG 2000 Part I Final Committee Draft Version 1.0," 2000, http://www.jpeg.org/public/fcd15444-1.pdf.

[5] G. Campbell, T. DeFanti, J. Frederiksen, S. Joyce, and L. Leske, "Two bit/pixel full color encoding," in *SIGGRAPH*, 1986, pp. 215-223.

[6] "Bump Map Compression," Nvidia Corporation, October 2004, http://download.nvidia.com/developer/Papers/2004/Bump_Map_Compression/Bump_Map_Compression.pdf.

[7] "3Dc White Paper", ATI Technologies, April 2004, http://www.ati.com/products/radeonx800/3DcWhitePaper.pdf.

[8] E. Delp and O. Mitchell, "Image compression using block truncation coding," in *IEEE Trans. Communications*, 1979, vol. COM-27, pp. 1335-1342.

[9] "Bump Map Compression", Nvidia Corporation, http://developer.nvidia.com/object/bump_map_compression.html.

[10] Crystal Space 3D, http://www.crystalspace3d.org.

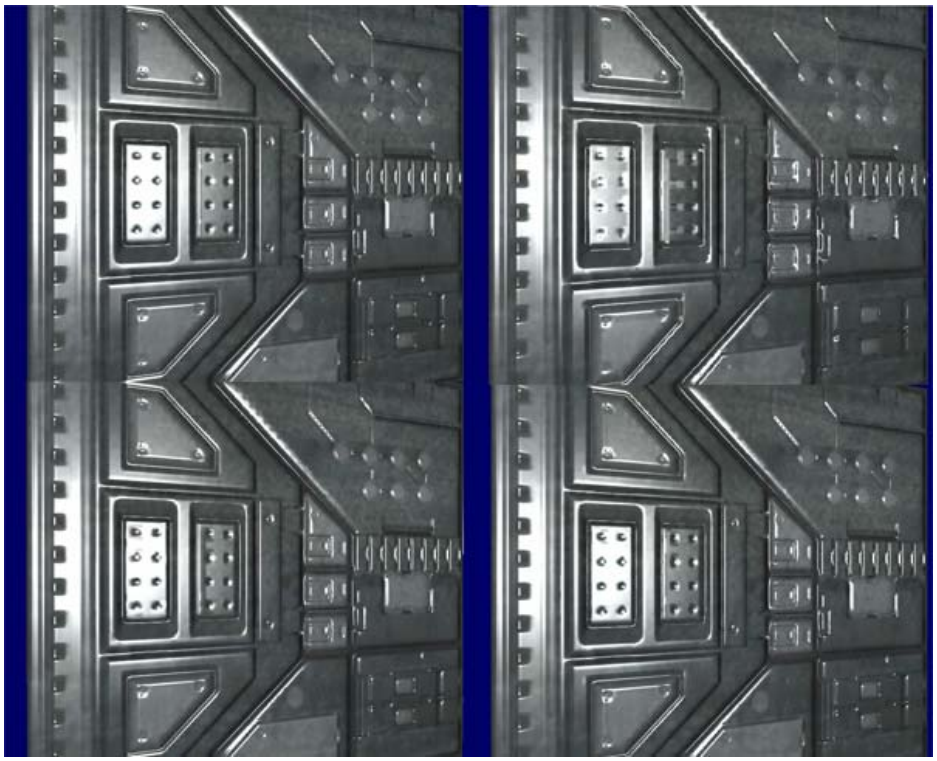

Figure 3: DOOR test case
Top-left: Uncompressed; Top-right: DXTC; Bottom-left: Proposed Method; Bottom-right: 3Dc

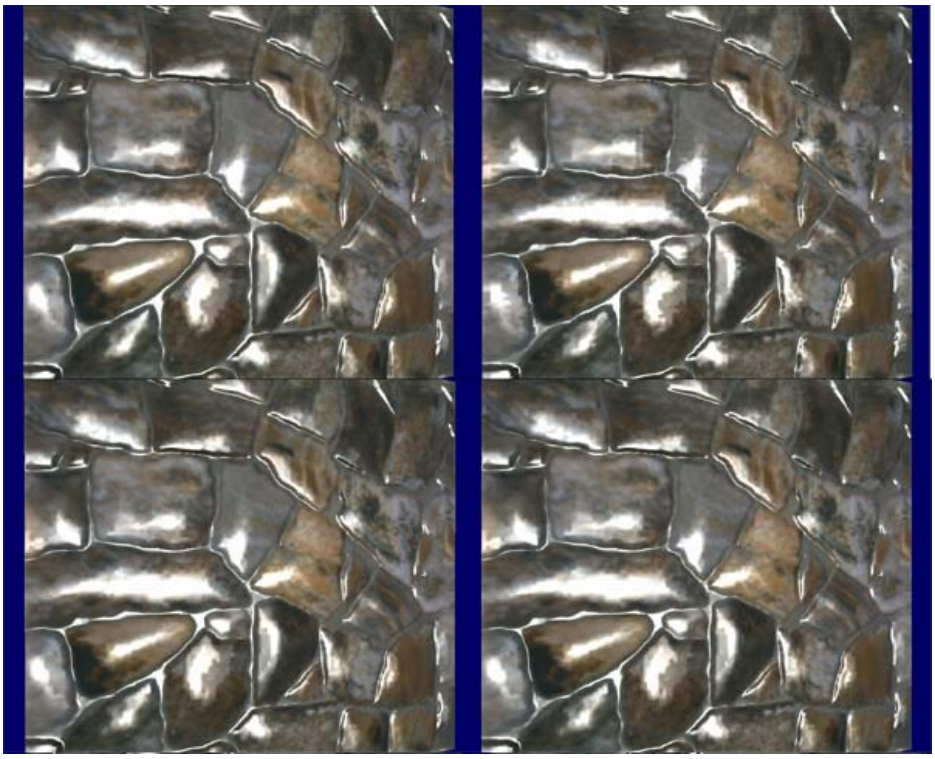

Figure 4: BRICK test case
Top-left: Uncompressed; Top-right: DXTC; Bottom-left: Proposed Method; Bottom-right: 3Dc

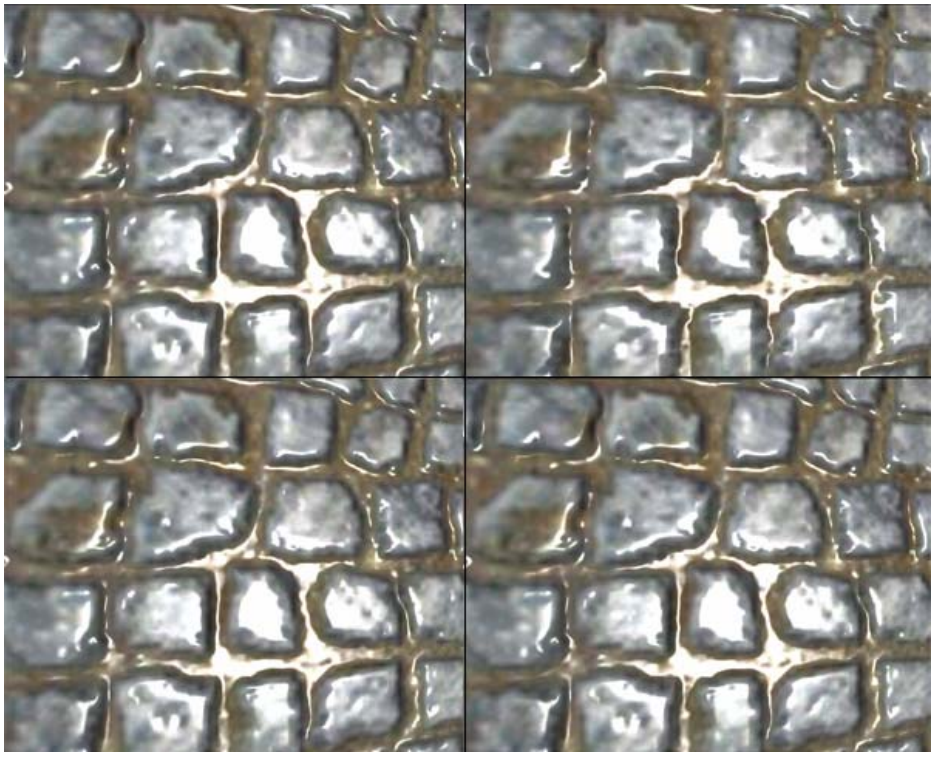

Figure 5: COBBLES test case
Top-left: Uncompressed; Top-right: DXTC; Bottom-left: Proposed Method; Bottom-right: 3Dc

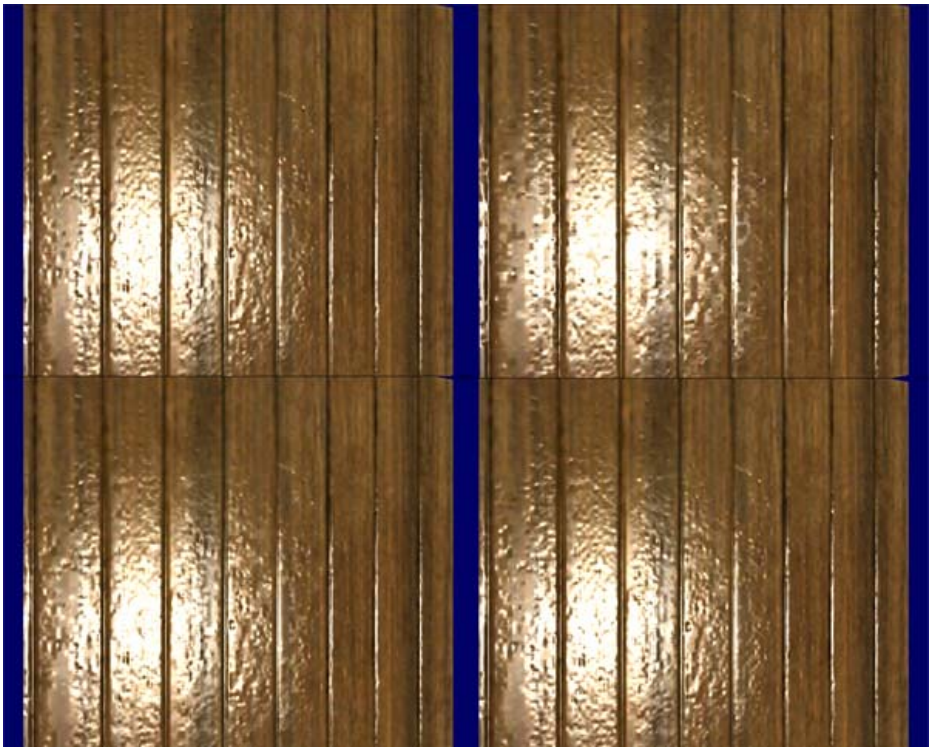

Figure 6: WOOD test case
Top-left: Uncompressed; Top-right: DXTC; Bottom-left: Proposed Method; Bottom-right: 3Dc