Compression of Synthesized Textures

F. Brayner
Centro de Informática-UFPE
Recife, Brazil
flb@cin.ufpe.br

M. Walter
Instituto de Informática-UFRGS
Porto Alegre, Brazil
marcelo.walter@inf.ufrgs.br

***Abstract*—In spite of graphics hardware advancements, graphics memory is still a scarce resource for usual applications. Besides, for most raster-based applications, the available bandwidth is one important limiting factor for increasing performance in the system. Texture compression addresses both of these problems. We introduce a new technique for compression of textures synthesized from samples. The compressed texture stores the sample plus encoded data, gathered during synthesis, which enables real-time decompression of large textures. With this scheme we are able to achieve high compression rates. Our solution explores a spectrum of textures where general texture compression schemes achieve less than optimal compression rates. These are usually textures with repeating patterns, regular or near-regular ones, and stochastic ones, the exactly types of textures where texture synthesis algorithms perform well. We also present analytical formulae for our compression scheme that allows an exact computation of compression rates achieved.**

***Keywords*-texture compression; texture synthesis**

I. Introduction

In countless applications, textures play an important role for achieving realism. Applications demand many high resolution textures and therefore memory space for storing all textures is a problem. Another related problem is the transmission of textures from memory to GPU. Memory bandwidth is generally the limiting factor for improving performance [1]. With texture compression, a compressed version of the texture – usually lossy – is stored and transmitted when needed. Decompression is done in execution time. Since compression is done offline and only once, compression techniques afford non-real time algorithms.

In spite of similarities, texture compression differs from image compression in some key points. When designing a texture compression scheme, we are not just concerned about storage and transmission. When it comes to rendering we have to take into account other aspects such as decoding speed and random access, for instance. These aspects are well-defined in [2]. Decode speed becomes even more important for real-time rendering applications. On the other hand, there are no strong requirements under encoding speed since it usually happens offline. Random access is mandatory since we usually don't know how the render system will access the texture.

We show in this paper how to use texture synthesis from samples to achieve texture compression. Texture synthesis from samples is a powerful idea. Pixels or patches from the sample are copied into the final desired texture, illustrated in Figure 1. Since all necessary information to build the final texture already exists in the sample, we can think of the synthesis process as finding the best patches to combine in the final output. Although texture synthesis algorithms are often too slow or complex for real time rendering applications on graphics hardware, we show how to tackle this issue by gathering information about the synthesis process and encoding it together with the input sample. Thus, our compressed texture stores not just the sample, but also information on how to build the synthesized result. This metadata is small and therefore a high level of compression is possible. The decompression algorithm simply rebuilds the result from the sample following the stored "recipe", allowing real-time decompression.

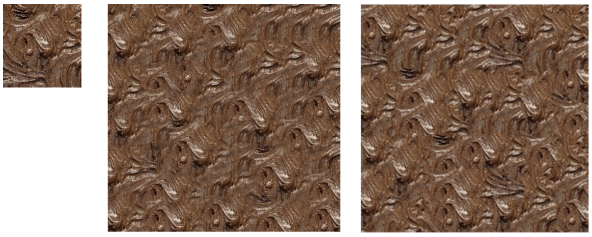

Figure 1. Example of Patch-Based Texture Synthesis. Sample on the left is 122×128. Both results are 512^2. The one in the middle used Liang's technique [4] and the one on the right used image quilting [5].

The potential of texture synthesis to achieve compression was also used by Wei and colleagues in a technique called *Inverse Texture Synthesis* [3]. The main difference is that in that work, the idea was to find a compressed exemplar for a collection of real-captured textures, including dynamic ones, with time-varying data. Our work shares the same motivation but with different goals.

II. Related Work

We review in this section both texture compression techniques and texture synthesis from samples, since our work brings together these two areas.

A. Texture Compression

Texture compression is an old graphics research topic. Already in 1979 Delp and Mitchell presented Block Truncation Coding (BTC), a simple lossy scheme for compressing images [6]. BTC compressed gray scale images in blocks of $4x4$ pixels. For each block, two gray scale colors are stored and each pixel is approximated by one of these two values. Therefore they were able to achieve 2 bits per pixel (bpp) compression rate. Their work was seminal and influenced later compression algorithms such as Color Cell Compression (CCC) [7] which allowed compression of colored textures at 2 bpp compression rate, and others such as [8], [9].

An important compression technique, that can be seen as a further adaptation of BTC/CCC methods, is the S3 Texture Compression (S3TC) [9], used in DirectX, where its known as DXTn depending on how it handles the alpha channel (DXT1, DXT3, and DXT5 are the most commonly used). In S3TC the textures are compressed by coding each $4x4$ pixel tile into a 64 bit data chunk. Two base colors in RGB565 format are stored in the first block's half. In the second block's half, a two-bit index is stored for each pixel. This index points to a local color set that consists of the two base colors and two additional colors in-between the base colors. The quality of S3TC is generally higher than that given by CCC but the gains are achieved at a 4 bpp cost. One disadvantage of this method is that only four colors can be used per block. Ivanov and Kuzmin [10] tried to solve this problem by using colors from neighboring blocks but the memory bandwidth increase became a drawback. Nevertheless, S3TC has become a leading texture compression method.

Used in many areas such as audio and image compression, Vector Quantization (VQ) also influenced several texture compression schemes. VQ tries to find a small number of vectors to approximate a given vector distribution keeping the error small. Beers et al. [2] use vector quantization for texture compression to reach high compression. The presented scheme allowed fast texture decoding achieving rates of 1-2 bpp. A simplified derivation of this research was also co-developed and implemented in Sega Dreamcast games console hardware. Others VQ based schemes were proposed [11] and [12]. However, the major drawbacks with these VQ based schemes are indirect data access and codebook handling. To retrieve a single texture element, two memory accesses are needed and the codebook size can be too expensive for implementation in hardware.

Fenney [14] presented a departure from BTC or VQ based schemes. His scheme relied on the fact that lowpass filtered signals are often good approximations of original signals. Each pixel can take its color from one of two bilinearly upscaled images or from two blended values between these two images. These two images are derived from the original texture and stored. The bilinear magnification happens at decompression time. His scheme is presented in two variations that can lead to 2bpp or 4bpp. This technique is being used under the name of PVR-TC in the PowerVR MBX technology.

In 2005 iPACKMAN/Ericsson Texture Compression (ETC) was presented at Graphics Hardware [15]. The iPackmann technique still uses the block-based ideas from BTC and works by building a fixed codebook and separates the luminance information from chrominance. For each $4x4$ block, two $2x4$ or $4x2$ sub-blocks are considered, each with one base color. The luminance values are refined with modifiers in a modifier table pointed by the per-texel indices. The algorithm has a 4bpp compression rate. The major problem with this scheme is the limitation of having only one chrominance per sub-block giving rise to block artifacts for some textures. More recently ETC2 [16] was presented to reduce the block artifacts in the previously ETC compression scheme. The paper shows how some invalid bit combinations can be useful to improve the ETC scheme. In general, the paper works on some specific encoded bit sequences (just discarded in the original work) to enable new modes of the algorithm, and these new modes try to improve where ETC performs poorly, such as blocks with two distinctly chrominances. The authors claim that the additional complexity added to the algorithm worth the quality improvement of 0.82dB.

Our proposal is a departure from the usual compression techniques, since it uses texture synthesis as the basic tool to compute a compressed texture.

B. Texture Synthesis from examples

The idea of using a real texture sample to drive results in texture synthesis has a long tradition in Computer Graphics and Image Processing. Early work on this topic is presented in [17], [18], [19], [20]. In 1993, Popat and Picard [21] presented a probabilistic model for texture synthesis and also suggested that their work could be used for texture compression.

Texture synthesis from samples is an excellent solution for building textures that are not only visually similar to the given sample but also can be built at user-defined resolutions. The advances in this area grew from pixel-based techniques [22], [23], to more recent patch-based techniques [4], [5], [24], [25], [26], where the final texture is formed by joining together pieces or blocks of the original sample, with a RGB metric for selecting the best matches. There has been a lot of activity in this area in the last few years and a good review of the work so far in this area is given in the Siggraph 2007 course presented by Kwatra and colleagues [27].

Using texture synthesis from samples in compression tasks is mentioned in [22] and also in [23]. This last work already used vector quantization to speed-up their solution,

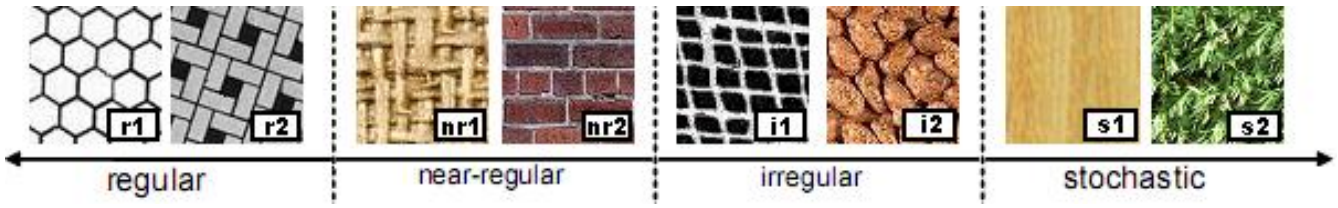

Figure 2. For the classes of textures in the above spectrum, synthesis algorithms can offer higher compression rates than most general texture compression techniques. These samples (r - regular, nr - near-regular, i - irregular and s - stochastic) were chosen as a texture set for comparison with other compression schemes in Table II. Figure adapted from [13].

a technique familiar in image/texture compression. In [28] there is mention in future work of using fractal techniques [29] combined with Image Analogies for compressing textures.

Li-Yi Wei explored the idea of trying to keep a small texture data to generate a larger virtual texture on [30]. His scheme first stores a set of texture tiles instead of a large texture, and then generates an arbitrarily large and non-periodic virtual texture map from the relatively small set of stored texture tiles. Our work follows a similar idea in the sense of keeping a small texture data to generate a larger texture. In our case, we focused on other sample-based texture synthesis than the tile-based ones. We do not need to store a set of tiles, just the texture sample used in the synthesis process, plus data that is minimum when compared with the sample texture data. We overcome the main issues of using texture synthesis as a texture compression scheme (as most texture synthesis techniques are often too slow or too complex for graphics hardware) by saving data about the synthesis process such that re-synthesizing is possible in real time.

We explored two alternative Patch-Based Texture Synthesis (PBTS) techniques to build the textures that we will compress later. We implemented the approach by Liang [4], and Image Quilting [5]. However, our solution works potentially with many PBTS techniques, since they all share the same basic principle of searching patches from the sample and combining them in a consistent way. A potential drawback of our solution could be the set of textures which we can correctly synthesize using synthesis from samples. Fortunately, a large number of textures, either natural or man-made, are already well handled by current solutions. We will show in the results section how our compression scheme works for the full spectrum of textures illustrated in Figure 2.

III. Our Approach

In general terms, our technique is very simple. While the texture is being built, using a synthesis from example algorithm, we capture and store the data needed to rebuild the result on graphics hardware. For the synthesis part we used a small variant of the work presented in [4] and [5]. Below we detail the process.

A. Synthesis Process

Our work uses the approach from [4] and [5] for PBTS. Both techniques share many similarities, and differ mainly on how the patches are blended. In basic PBTS, patches of the original sample are combined to form the final texture. The algorithm starts by randomly picking a patch B_0 to start the process. This patch is positioned at the bottom left corner of the output texture (Figure 3 (left)). The size w_B of the patches is user-defined and intuitively it should be the size of the main structures present in the sample. For most textures, using a patch of size between half and a quarter of the size of the original sample works well. For simplicity they are also restricted to square patches. The synthesis process follows by adding patches side-by-side and once a full row is completed the process continues for the row above and so on – Figure 3 (middle and right).

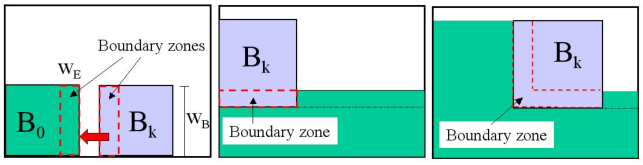

Figure 3. Illustration of Patch-Based Texture Synthesis adapted from [4]. Starting point (first configuration), second configuration for patch matching, and third configuration (L-shaped matching). Green areas represent texture already synthesized.

For each patch, there is a boundary zone with a user-defined width w_E. There are three possible configurations for boundary zone matching, as illustrated in the same figure: vertical, horizontal, and L-shaped. The optimal size of w_E depends on the texture being generated. If it is too small, it will not capture enough details. If it is too large it will negatively impact the algorithm's performance. As a balance w_E is typically set as $\frac{1}{6}$ of w_B. The values set for w_B and w_E are very important, since they ultimately define the initial conditions for the synthesis process. Later extensions on texture synthesis from patches have touched on this issue, using different patch sizes computed adaptively [24][25].

The critical part of the algorithm is the selection of the next patch B_k to be pasted onto the texture being constructed. As with many texture from sample techniques before [22][23], a list of candidate patches which satisfy an

error criterion is built. The error is measured using an $L2$ norm on the RGB channels. From this list a random patch is selected. To build this list, the input sample is searched for all possible patches. If there is no patch satisfying the condition, the algorithm picks the patch with the smallest distance. Usually, all possible patches in the sample are searched, but building a list with a fixed number of patches randomly chosen also works.

More formally, given two texture patches I_1 and I_2 of the same size and shape, they match if

$$d(I_1, I_2) < \delta = \tau \left[\frac{1}{A} \sum_{j=1}^{A} \left(p_{E_i}^j \right)^2 \right]^{1/2}$$

where A is the number of pixels in the boundary zone, $p_{E_i}^j$ represent the values of the j^{th} pixel in the E_i boundary zone, $d()$ represents the distance between the two patches, and τ is a defined constant (default value is $\tau = 0.2$). This distance is computed only for the boundary zone E of patches as follows:

$$d(E_i, E_{i+1}) = \left[\frac{1}{A} \sum_{j=1}^{A} \left(p_{E_i}^j - p_{E_{i+1}}^j \right)^2 \right]^{1/2}$$

Once the patches are selected, there is a blending step to provide smooth transition among adjacent patches. This is a crucial difference between the two techniques. In the work by Liang [4] the smoothing is performed with feathering, or simply linear interpolation in the overlapping area, as proposed by Szeliski and Shum [31]. In the image quilting approach, the transition is performed with a minimum error boundary cut technique. Basically, in a given row of the overlapping area, the mincut searches for the pixel in the next row that minimizes the error, restricted to three possible cases, illustrated in Figure 4.

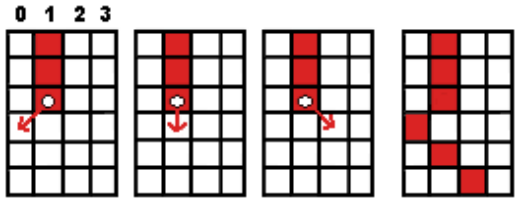

Figure 4. Mincut cases and a complete cut example

The main adaptation when using the above algorithm for texture compression is that we have to keep track of which patches are being selected to build the final texture. Since the final result is a collection of patches from the sample, properly combined, we need to store three types of information: fixed-sized information about the synthesis process, that is, the sizes of w_E and w_B; the coordinates which define where the selected patches start, plus information on how to blend the patches. These last two are variable-sized, since the amount of information in bits will vary depending upon the desired result, and on the type of blending used.

We store, for each patch of the output, a pair (x, y) of coordinates which define the upper-left corner of the patch in the space of the sample. We will call the set of all pairs of coordinates ψ. If we know offhand these coordinates, for all patches that make up the whole result, we are able to rebuild the texture instantaneously. This is the basis of our compression algorithm. Since almost all samples used in typical synthesis algorithms are small, under 256 pixels in each dimension, storing the coordinates will take at most two bytes for each patch. We have to store the values of w_B and w_E as well, since they are needed to completely define the patch. If we are rebuilding the texture with the Liang's technique this is all information needed; on the other hand, if we are using image quilting, we also have to store the pixels coordinates, which define the mincut boundary.

IV. Realtime Decompression on GPU

Our compression scheme allows real-time decompression. It can be done on the GPU or simply in software. This is possible since we have all selected patches coordinates encoded in the compressed file, allowing us to jump the most expensive step in the texture re-synthesis (decompression). For decompression we have as input information the original sample plus a set ψ of pairs of coordinates, which define where each patch starts. We also have w_E and w_B as explained above, and for image quilting we also have the coordinates of the mincut, for each pair of neighboring patches.

Through the normalized texture coordinates and the selected patches coordinates, with simple calculations we get the right offsets and, finally, the final pixel color. With the texture coordinates and the final texture resolution we can easily find out the pixel location in the "virtually" decompressed texture. Note that the random access requirement is easily achieved since for any random texture coordinates we get the right offsets and reach the final pixel color.

The metadata containing all the selected patches coordinates is accessed in the pixel shader through a lookup table. In fact, we have coded these coordinates as RGBA data, that is, a 4 byte word RGBA encodes information for two patches. The RGBA format is convenient because it reduces the number of texture fetches. For instance, in a decompression process with four patches (Fig. 5) in the final texture we would have the following coordinates data/texture data:

- Coordinates for 4 patches:
  (x_{b0},y_{b0}) (x_{b1},y_{b1}) (x_{b2},y_{b2}) (x_{b3},y_{b3})
- Equivalent one-dimensional texture data:
  (R, G, B, A, R, G, B, A)

We also store the mincut coordinates in RGBA words, as follows. For each row of pixels in the mincut region, we

store the number of pixels there are from the left border to the cut. For instance, for the 6 rows of pixels in Figure 4, we would have the following stored:

((1)R, (1)G, (1)B, (0)A, (1)R, (2)G, B, A)

If w_B is not multiple of 4 we waste the few remaining bytes from the last word, as in the above example, where the bytes B and A would have no data stored.

In both compression variants we are using in this paper, a given texel will be found in one of three possibilities in the "virtually" decompressed texture. In Figure 5 we illustrate these cases: green area, where the final color can be obtained directly from the sample; yellow area, where two patches are involved in the pixel synthesis and red area, where four patches are involved in the pixel synthesis. After obtaining the coordinates of the patches involved in synthesizing a given texel from the lookup table, the next step depends on the texture synthesis approach selected to compress the texture. In the next subsections we have more specific details of each approach.

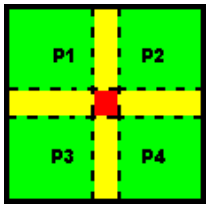

Figure 5. A four patches synthesized texture. In the decompression process texels can fall in one of three cases, represented by the 3 colors.

A. Liang based decompression

In Liang based decompression, when the texel falls outside the overlapping area (green area in Figure 5), we can get the final color directly from the sample without performing any operation over it. This access in the sample is done through its respective selected patches coordinates.

In the second case, when the texel falls inside an overlapping area (yellow and red areas in Figure 5), we get the coordinates of all patches involved in the overlapping and calculate the interpolations, a single linear interpolation for the yellow area and three linear interpolations in the red area. For instance, if it is a simple vertical/horizontal overlapping (as shown in Figure 3 left), we get the coordinates of the two patches adjacent to the desired texel and use them to generate the correct final texel color with a single interpolation. This final texel will have exactly the same color when it was first synthesized (compressed).

For the L-shaped overlapping, most pixels are solved as described above. A special case occurs where the vertical and horizontal overlapping regions meet (red area). In this case we need the coordinates of the four patches that generated the color of the pixels in this region. With this four patches coordinates we get the final color with three linear interpolations.

B. Image Quilting Based Decompression

As in the Liang scheme, the texels will fall inside or outside the overlapping area, the difference here is that texels inside the overlapping area are computed with mincut. After obtaining the patches involved in the synthesis process, we need to decide from which patch it came from. This selection is easily done once we have the mincuts encoded in the compressed format.

For a simple vertical overlap we just need to know whether the pixel is left or right of the cut. This is done with a single access to the encoded data lookup table. The horizontal overlap is analogue to the vertical one. For the L-shaped overlapping three cuts are considered, so we know the pixel location with a minimum of one and a maximum of three accesses to the lookup table.

As we can see in Fig. 5, the area where the decompression is more expensive is relatively small (red area) compared to the total size of the texture. The relative red area is even smaller for outputs with higher resolutions. This is the key to maintain the frame rate even with some extra lookup table access.

V. Results

We want to assess how much compression is possible with our scheme. The efficiency of texture compression algorithms can be measured using the number of bits per pixel (bpp) needed to store the compressed texture. In Table I we define the variables we use. For simplicity, we will consider square output textures with k patches in either dimension; k^2 is the total number of patches needed to synthesize the final output texture and at least $k = 2$. In this case $m_{out} = n_{out}$.

Table I
Names and Meanings of Variables.

Name	Meaning
m	x-resolution of sample
n	y-resolution of sample
m_{out}	x-resolution of the result
n_{out}	y-resolution of the result
k^2	total number of patches
w_e	overlap region among patches (in # of pixels)
w_b	patch width (in # of pixels)

The resolution of the final texture is defined as a function of number of patches k, and sizes w_B, and w_E as follows:

$$(kw_B - (k-1)w_E)^2$$

We will derive the compression rate for each one of the texture synthesis algorithms we tested: Liang [32] and Image Quilting [5].

A. Liang

The amount of compression is given by:

$$bpp = \frac{8 \cdot 3 \cdot mn + c}{m_{out}^2}$$

where $24mn$ is the cost of storing the sample and c is a function to measure the extra cost of storing the information to rebuild the texture. For each k_i patch we have 2 bytes for storage of the pair of coordinates plus 8 more bytes fixed for storing header information, that is, the sizes of w_E and w_B. Therefore $c = 8(2k^2 + 8)$ and the equation becomes:

$$bpp = \frac{24mn + 16k^2 + 64}{m_{out}^2}$$

Since the output resolution is given indirectly by the number of patches and their size, we have $m_{out} = kw_B - (k-1)w_E$.

B. Image Quilting

The amount of compression is given by the same equation as in the Liang case, except that here we have the extra cost of storing, for each overlapping area, the cuts offsets. For each cut we need w_B entries. The number of bits needed for each entry is a function of w_E, since we need $\lceil \log_2 w_E \rceil$ bits for storing this information.

For example, in Fig. 4, the complete path should be stored as (1, 1, 1, 0, 1, 2), so the maximum offset would be w_E. For the L-shaped overlapping area, we consider the 'L' cut as two separated cuts, a vertical one and an horizontal one.

$$bpp = \frac{24mn + 16k^2 + 64 + 2k(k-1)w_B \lceil \log_2 w_E \rceil}{m_{out}^2}$$

C. Numerical Examples

A numerical example might help understand these formulae. Let us say we have $k = 11, w_B = 36, w_E = 14, m = n = 64$. In this case $m_{out} = 256$, and the final *bpp* for each synthesis algorithm is:

- Liang

$$bpp = \frac{24 \cdot 64 \cdot 64 + 16 \cdot 121 + 64}{256^2} = \frac{100304}{65536} = 1.53$$

- Image Quilting

$$bpp = \frac{24 \cdot 64 \cdot 64 + 16 \cdot 121 + 64 + 2 \cdot 11 \cdot 10 \cdot 36 \cdot 4}{256^2}$$

$$bpp = \frac{100304 + 31680}{65536} = 2.0$$

In other words, a 256^2 texture, synthesized from a 64^2 sample, can be stored using our scheme with at most 2.0 *bpp*. Larger output textures will have an even better rate of compression since the fixed cost is amortized by a larger output. For instance, if we make $k = 20$, the output texture resolution is 454^2 and we have, for the worse case (image quilting):

$$bpp = \frac{24 \cdot 64 \cdot 64 + 16 \cdot 400 + 64 + 2 \cdot 20 \cdot 19 \cdot 36 \cdot 4}{454^2}$$

$$bpp = \frac{214224}{206116} = 1.0$$

In Figure 6 we plot how the rate of compression in bits per pixel behaves with the increase in the number of patches k, for image quilting, keeping all other parameters fixed. Higher k results in better compression rates.

In Table II we have some compression results for all textures in Figure 2. Note that most compression rates are hardly reached by general texture compression algorithms. The In/Out column refers to sample resolution/final texture resolution. Another aspect that we should mention is that our texture compression scheme is lossless as far as texture synthesis itself is lossless, that is, our solution does not introduce further degradation. Figure 7 shows how other texture compression schemes degrade the visual quality of the texture against our scheme.

Table II
COMPRESSION RESULTS.

Texture	In/Out	Liang bpp	Quil. bpp
$r1$	74x75/256^2	2,04	2,55
$r1$	74x75/512^2	0,52	1,10
$r2$	64x64/256^2	1,52	1,83
$r2$	64x64/512^2	0,40	0,97
$nr1$	64x64/256^2	1,53	2,01
$nr1$	64x64/512^2	0,40	0,97
$nr2$	64x64/256^2	1,52	1,83
$nr2$	64x64/512^2	0,40	0,97
$i1$	90x90/256^2	2,98	3,30
$i1$	90x90/512^2	0,76	1,34
$i2$	80x80/256^2	2,37	2,86
$i2$	80x80/512^2	0,61	1,18
$s1$	60x60/256^2	1,35	1,83
$s1$	60x60/512^2	0,35	0,93
$s2$	60x60/256^2	1,35	1,83
$s2$	60x60/512^2	0,35	0,93

VI. CONCLUSIONS AND FUTURE WORK

We presented a texture compression algorithm which uses texture synthesis from samples as the main tool for compression. Our solution is lossless in the sense that it does not introduce artifacts, except the ones inherent to texture synthesis. During the synthesis step, we gather information on which patches from the sample are being used to form the final output texture. The compressed texture is the sample plus the data gathered during the synthesis step. This information is efficiently encoded on a 1D texture. The whole solution was implemented on GPU.

Texture synthesis was considered to be prohibitive for being used as the core of a texture compression scheme. We showed how to overcome the main issues and presented

Figure 7. One advantage of our texture compression scheme is that it does not further degrade the results. Here we show the quality of several texture compression schemes on the literature. Images were generated by PVRTexTool, a tool that comes with PowerVR SDK.

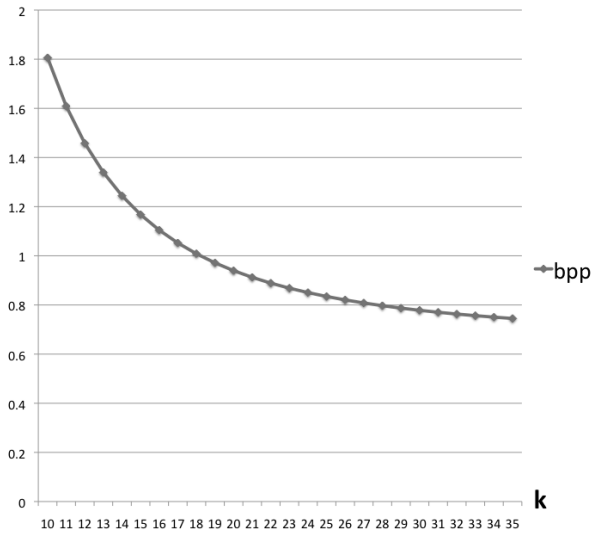

Figure 6. Rate of compression (bpp) against k (number of patches) for Image Quilting based texture compression. When k increases so does the output resolution and better compression rates are achieved. Other values as $w_B = 44$, $w_E = 18$, $m = n = 64$.

a solution which uses texture synthesis as a simple texture compression scheme running in real-time. Our solution leads to compression rates, in several cases, never reached by other compression schemes. In Table II we showed several compression cases. We demonstrated our technique with two popular PBTS techniques, the one due to Liang and colleagues [32] and Image Quilting [5].

One possible limitation of our solution is that we can only compress textures which can be built with PBTS techniques. This is in fact not a serious limitation since PBTS techniques are capable of handling a large number of textures, as exemplified in the set of textures presented in Figure 2. These textures are not optimally compressed by the general texture compression schemes in the literature, as noted in [30].

We are currently investigating how our approach could be used to achieve even better compression rates. Similar to the work presented in *Inverse Texture Synthesis* [3], we realized that only a fraction of the sample is needed for the synthesis process. In other words, we do not need to store the whole sample to rebuild the texture, but only the needed pixels. As a first step towards this goal we investigated, for a few cases, how much of the sample is being used during the synthesis process. Figure 8 illustrates the idea. The better case was for the texture shown, where only 71% of the original sample was needed to build the result. We are currently pursuing further investigations to better explore this possibility.

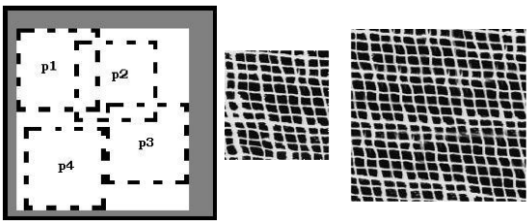

Figure 8. Illustration of how we can optimize storage of the sample. The gray area was not used during synthesis and therefore does not need to be stored. Sample on the middle is 126^2 and result is 190^2. Amount of sample used = 71%.

ACKNOWLEDGMENT

Work partially supported by FACEPE through grant APQ-0203-1.03/06 and CNPq through grant 483356/2007.

REFERENCES

[1] T. Aila, V. Miettinen, and P. Nordlund, "Delay streams for graphics hardware," *ACM Transactions on Graphics*, vol. 22, pp. 792–800, 2003.

[2] A. C. Beers, M. Agrawala, and N. Chaddha, "Rendering from compressed textures," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1996, pp. 373–378.

[3] L.-Y. Wei, J. Han, K. Zhou, H. Bao, B. Guo, and H.-Y. Shum, "Inverse texture synthesis," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 52:1–52:9, Aug. 2008.

[4] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum, "Real-time texture synthesis by patch-based sampling," *ACM Transactions on Graphics*, vol. 20, no. 3, pp. 127–150, July 2001.

[5] A. Efros and W. Freeman, "Image quilting for texture synthesis and transfer," *Proceedings of SIGGRAPH 2001*, pp. 341–346, August 2001, iSBN 1-58113-292-1.

[6] M. O. Delp E., "Image compression using block truncation coding," *IEEE Transactions on Communications*, pp. 1335–1342, 1979.

[7] G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, and L. A. Leske, "Two bit/pixel full color encoding," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 215–223, 1986.

[8] G. Knittel, A. G. Schilling, A. Kugler, and W. Straßer, "Hardware for superior texture performance," *Computers & Graphics*, vol. 20, no. 4, pp. 475–481, Jul. 1996.

[9] O. Konstantine, N. Krishna, and H. Zhou, "System and method for fixed-rate block-based image compression with inferred pixel values," 1999.

[10] D. V. Ivanov and Y. Kuzmin, "Color distribution - a new approach to texture compression," *Comput. Graph. Forum*, vol. 19, no. 3, 2000.

[11] Y. su Kwon, I. cheol Park, and C. min Kyung, "Pyramid texture compression and decompression using interpolative vector quantization," in *In ICIP*, 2000, pp. 89–106.

[12] Y. Tang, H. Zhang, Q. Wang, and H. Bao, "Importance-driven texture encoding based on samples," *cgi*, vol. 0, pp. 169–176, 2005.

[13] W.-C. Lin, J. H. Hays, C. Wu, V. Kwatra, and Y. Liu, "A comparison study of four texture synthesis algorithms on regular and near-regular textures," Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-04-01, January 2004.

[14] S. Fenney, "Texture compression using low-frequency signal modulation," in *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 84–91.

[15] J. Ström and T. Akenine-Möller, "ipackman: high-quality, low-complexity texture compression for mobile phones," in *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. New York, NY, USA: ACM Press, 2005, pp. 63–70.

[16] J. Ström and M. Pettersson, "Etc2: texture compression using invalid combinations," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 49–54.

[17] K. S. Fu and S. Y. Lu, "Computer generation of texture using a syntactic approach," *Computer Graphics (SIGGRAPH '78 Proceedings)*, vol. 12, no. 3, pp. 147–152, Aug. 1978.

[18] J. Monne, F. Schmitt, and D. Massaloux, "Bidimensional texture synthesis by markov chains," *Computer Graphics and Image Processing*, vol. 17, no. 1, pp. 1–23, Sep. 1981.

[19] G. Cross and A. K. Jain, "Markov random field texture models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 5, no. 1, pp. 25–39, Jan. 1983.

[20] A. Gagalowicz and S. Ma, "Model driven synthesis of natural textures for 3-D scenes," in *Eurographics '85*, 1985, pp. 91–108.

[21] K. Popat and R. W. Picard, "Novel cluster-based probability model for texture synthesis, classification, and compression," in *Proceedings SPIE visual Communications and Image Processing '93, Boston*, 1993, pp. 756–768.

[22] A. Efros and T. Leung, "Texture synthesis by non-parametric sampling," in *International Conference on Computer Vision*, vol. 2, 1999, pp. 1033–1038.

[23] L.-Y. Wei and M. Levoy, "Fast texture synthesis using tree-structured vector quantization," *Proceedings of SIGGRAPH 2000*, pp. 479–488, July 2000, iSBN 1-58113-208-5.

[24] A. Nealen and M. Alexa, "Hybrid texture synthesis," in *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, Jun. 2003, pp. 97–105.

[25] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, "Graphcut textures: Image and video synthesis using graph cuts," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 277–286, Jul. 2003.

[26] Q. Wu and Y. Yu, "Feature matching and deformation for texture synthesis," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 364–367, Aug. 2004.

[27] V. Kwatra, S. Lefebvre, G. Turk, and L. Wei, "Example-based texture synthesis." [Online]. Available: http://www.cs.unc.edu/~kwatra/SIG07_TextureSynthesis/coursenotes.htm

[28] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, "Image analogies," in *SIGGRAPH '01*. New York, NY, USA: ACM Press, 2001, pp. 327–340.

[29] M. F. Barnsley and L. P. Hurd, "Fractal modelling of real world images," in *Fractal Image Compression*, 1993, pp. 219–239.

[30] L.-Y. Wei, "Tile-based texture mapping on graphics hardware," in *Graphics Hardware 2004*, Aug. 2004, pp. 55–64.

[31] R. Szeliski and H.-Y. Shum, "Creating full view panoramic image mosaics and environment maps," in *Siggraph'97*, 1997, pp. 251–258.

[32] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum, "Real-time texture synthesis by patch-based sampling," *ACM Trans. Graph.*, vol. 20, no. 3, pp. 127–150, 2001.