Journal of Computer Graphics Techniques Vol. 1, No. 1
http://jcgt.org 2012

The Compact YCoCg Frame Buffer

Pavlos Mavridis Georgios Papaioannou
Department of Informatics, Athens University of Economics & Business

Figure 1. Our method can rasterize a color image using only two frame-buffer channels by interleaving the chrominance components in a checkerboard pattern. The final image is reconstructed using an edge-directed demosaicing filter. The compression error, visualized in the inset, is negligible, and the filtering is temporally stable.

Abstract

In this article we present a lossy frame-buffer compression format, suitable for existing commodity GPUs and APIs. Our compression scheme allows a full-color image to be directly rasterized using only two color channels at each pixel, instead of three, thus reducing both the consumed storage space and bandwidth during the rendering process. Exploiting the fact that the human visual system is more sensitive to fine spatial variations of luminance than of chrominance, the rasterizer generates fragments in the YCoCg color space and directly stores the chrominance channels at a lower resolution using a mosaic pattern. When reading from the buffer, a simple and efficient edge-directed reconstruction filter provides a very precise estimation of the original uncompressed values. We demonstrate that the quality loss from our method is negligible, while the bandwidth reduction results in a sizable increase in the fill rate of the GPU rasterizer.

1. Introduction

For years the computational power of graphics hardware has grown at a faster rate than the available memory bandwidth [Owens 2005]. This trend is common in computer hardware and is likely to continue in the future, making bandwidth-saving algorithms increasingly important. In addition to memory bandwidth, the storage space of GPUs is also limited; therefore, a reduction in the consumption of both resources is always desirable in real-time rendering.

The frame buffer, the area of memory that stores the resulting fragments during rasterization, is a big consumer of both memory bandwidth and storage space. The consumption of these resources is further increased by several factors, such as the usage of high-precision floating-point render targets, needed for high dynamic range (HDR) rendering and most importantly, by the usage of multisample render buffers, required for multisample antialiasing. A multisample frame buffer with N samples per pixel consumes N times the storage and bandwidth of a regular frame buffer, putting a lot of stress on the memory subsystem of the GPU.

This fact was recognized by hardware vendors, and many, if not all, of the GPUs shipping today employ proprietary lossless frame-buffer compression algorithms, that mostly exploit the fact that a fragment shader can be executed only once per covered primitive and that the same color can be assigned to many sub-pixel samples. According to information theory, there is no lossless compression algorithm that can guarantee a fixed-rate encoding; therefore, in order to provide fast random access, these algorithms can save only bandwidth but not storage.

In this article, we present a method to rasterize a color image using only two spatially interleaved color channels, instead of three, thus reducing the storage and more importantly, the bandwidth requirements of the rasterization process—a fundamental operation in computer graphics. This reduction in the memory footprint can be valuable when implementing various rendering pipelines. Our method is compatible with both forward and deferred rendering, it does not affect the effectiveness of any lossless compression by the hardware, and can be used with other lossy schemes, like the recently proposed SBAA [Salvi and Vidimče 2012], to further decrease the total storage and bandwidth consumption. Source code and a live WebGL demonstration of our method are available online in the supplemental materials for this article.

2. The YCoCg Color Space

The human visual system is more sensitive to spatial details in luminance than in chrominance. Therefore, an image-coding system can be optimized by encoding the chrominance components of an image with lower spatial resolution than the luminance ones, a process that is commonly referred to as chroma subsampling. This was exploited by many popular image- and video-compression algorithms, like JPEG and MPEG, and was also employed in television broadcasting for more than half a century.

The RGB to YCoCg transform decomposes a color image to luminance and chrominance components. This transform was first introduced in H.264 compression and has been shown to have better decorrelation properties than YCbCr or other similar transforms [Malvar and Sullivan 2003]. The transform is given by the following equation:

$$\begin{bmatrix} Y \\ C_o \\ C_g \end{bmatrix} = \begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 0 & -1/2 \\ -1/4 & 1/2 & -1/4 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{1}$$

This transform introduces some rounding errors when the same precision is used for the YCoCg and RGB data. In particular, converting the images of the Kodak image suite to YCoCg and back, results in an average PSNR of 52.1dB. This loss of precision is insignificant for our purpose and cannot be perceived by the human visual system. Still, this measurement indicates an upper limit in the quality of our compression scheme. We could avoid these rounding errors by multiplying the direct transform matrix in Equation (1) above by four. However, that would generate color components whose representation would require two more bits of precision compared to the RGB components. As described by Malvar and Sullivan, we can reduce this to one additional bit for each chrominance component by applying some well-known S-transform concepts, but that formulation, known as YCoCg-R, adds some additional overhead in the shader that is not justified in our case by a visible increase in quality. Thus, we decided to use the transform in Equation (1).

This transform has been performed traditionally in gamma-corrected (non-radiometric) color space for image and video compression. In this article, we have also used gamma-corrected sRGB data for our experiments, but, in the future, it would be very interesting to investigate how this transform performs in linear (radiometric) data and whether a different transform would perform better in this case.

3. Storage Format

The frame buffer in our scheme stores the color of the rasterized fragments in the YCoCg color space using two color channels. The first channel stores the luminance (Y) in every pixel. The second channel stores either the offset orange (Co) or offset green (Cg) of the input fragments, forming a checkerboard pattern, as illustrated in Figure 1. This particular arrangement corresponds to a luminance to chrominance ratio of 2:1 in each dimension and provides a 3:2 compression ratio, since two color channels are used instead of three. The same luminance to chrominance ratio is used by many video-compression codecs, commonly referred to as 4:2:2, but, in this case, the details on how the chrominance samples are produced and stored are different.

In order to create a compressed frame buffer on the GPU, the application can either request a render buffer with just two color channels, such as GL_RG16F or any similar format available in graphics APIs, or use a more traditional format with four color channels and use the free channels to store additional data. It is worth mentioning that some hardware support for chroma subsampling exists, in the form of the DXGI_FORMAT_R8G8_B8G8_UNORM and several other relevant texture formats, but to our knowledge these formats are not valid render targets, and they do not solve the same problems as the proposed technique.

The fragments produced by the rasterization phase of the rendering pipeline should be produced directly in the correct interleaved format. This is possible by making a simple modification of the fragment shaders used to render the scene, as shown in Listing 1. The final fragment color is converted to the YCoCg color space and, depending on the coordinates of the destination pixel, the YCo or YCg channels are emitted to the frame buffer.

Our method effectively downsamples the two chrominance channels using point sampling. Ideally, we would like to perform this downsampling using a

```
//convert the output color to the YCoCg space
vec3 YCoGg = RGB2YCoCg (finalColor.rgb);
ivec2 crd = gl_FragCoord.xy;
//store the YCo and YCg in a checkerboard pattern
finalColor.rg=((crd.x&1)==(crd.y&1))? YCoCg.rg:YCoCg.rb;
```

Listing 1. The GLSL code needed to directly produce fragments on the compact frame-buffer format.

decent resampling filter, like lanczos, in order to avoid any aliasing artifacts in the chrominance components of the frame buffer. However, such filtering is only possible when performing the downsampling as a post-processing operation, but not when it is performed on-the-fly during rasterization, as in our method. In practice, we did not observe any severe aliasing artifacts in the chrominance components from this lack of a presampling filter when testing with frame buffers from typical games.

4. Reconstruction Filters

When accessing the values of the compressed frame buffer, any missing chrominance information should be reconstructed from the neighboring pixels. The simplest way to do that is by replacing the missing chrominance value with the one from the *nearest* neighbor (Figure 2). This crude approximation shifts some of the chrominance values by one pixel, producing mosaic patterns at strong chrominance transitions, as shown in the red flag close-up of Figure 3. Using *bilinear* interpolation of the missing data from the four neighboring pixels mitigates these artifacts but does not completely remove them. These artifacts are not easily detectable by the human visual system in still images, since the luminance is always correct, but they can become more pronounced when motion is involved.

To eliminate these reconstruction artifacts, we have designed a simple and efficient *edge-directed* filter, where the weights of the four nearest chrominance samples are calculated based on the luminance gradient towards that sample. If the gradient has a value greater than a specific threshold, indicating an edge, then the corresponding chrominance sample has zero weight; other-

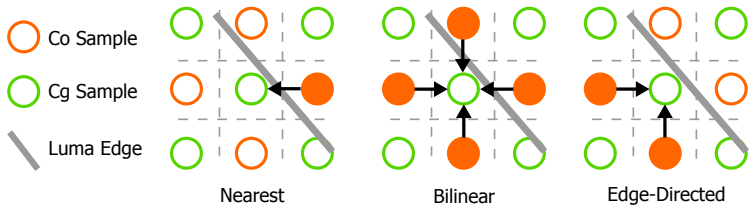

Figure 2. The edge-directed filter avoids sampling chrominance values beyond the edge of the current surface, leading to better reconstruction quality.

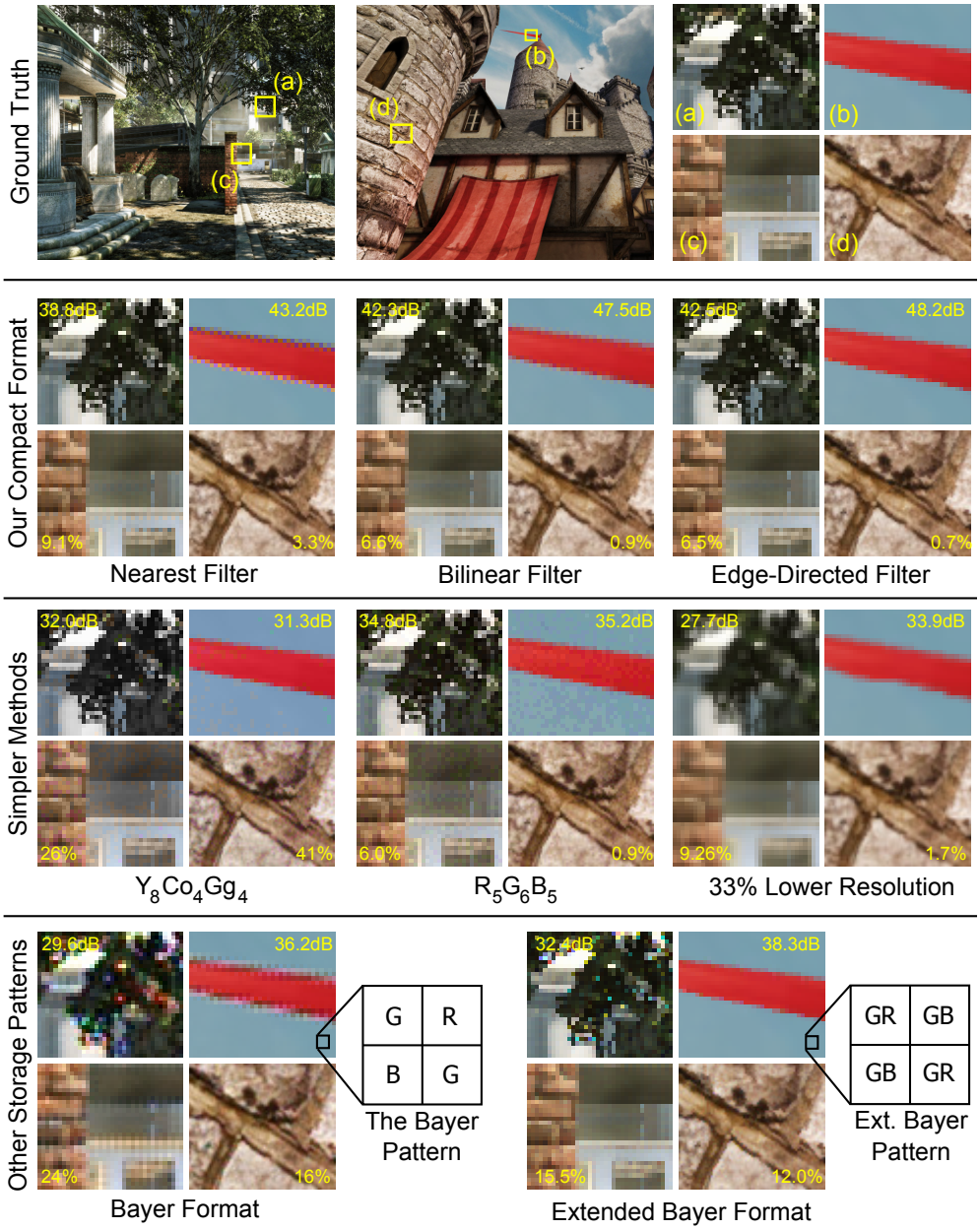

Figure 3. First Row: Uncompressed 8-bit per channel frame buffers in sRGB color space, used as input for our tests. Second row: Close-ups demonstrating our technique when using various reconstruction filters (Section 4). Third Row: Simpler methods (Section 5). Fourth row: Alternative mosaic patterns (Section 5). We report the PSNR and the perceptual difference [Yee 2004] compared to the original frame buffers. The best quality is achieved with our technique when using the edge-directed filter.

wise the weight is one. This is expressed compactly in the following equation:

$$C_0 = \sum_{i=1}^{4} w_i C_i, \qquad w_i = 1.0 - \text{step}(T - \mid L_i - L_0 \mid),$$

where C_i and L_i, respectively, are the chrominance (Co or Cg) and luminance of pixel i. Zero denotes the center pixel, while the next four values denote the four neighbors. The value T is the gradient threshold, which was set at 30/255 in our experiments. The step function returns one on positive values and zero otherwise. The gradient is computed as a simple horizontal and vertical difference of the luminance values, as shown in Listing 2. Our implementation does not use the built-in horizontal and vertical derivative functions (dFdx and dFdy), because on existing hardware, these functions return the same value for pixel blocks of 2×1 and 1×2, respectively; therefore, they cannot accurately capture the per-pixel variations of the frame-buffer values. In the special case, where all the weights are zero, the weight of the first neighbor is set to one. Furthermore, to avoid handling a special case at the edges of the frame buffer, where only pixels inside the frame boundaries should be considered, we are using a "mirrored repeat" wrapping mode when sampling the frame-buffer pixels, which is supported natively by the texture hardware. It is worth noting that the implementation of this filter uses conditional assignments that are significantly faster than branches on most architectures.

The design of the edge-directed filter is based on the observation that, while the luminance and chrominance channels are uncorrelated to an extent

```
//Returns the missing chrominance (Co or Cg) of a pixel.
//a1-a4 are the 4 neighbors of the center pixel a0.
float filter(vec2 a0, vec2 a1, vec2 a2, vec2 a3, vec2 a4)
{
  vec4 lum = vec4(a1.x, a2.x , a3.x, a4.x);
  vec4 w = 1.0-step(THRESH, abs(lum - a0.x));
  float W = w.x + w.y + w.z + w.w;
  //handle the special case where all the weights are zero
  w.x = (W==0.0)? 1.0:w.x;  W = (W==0.0)? 1.0:W;
  return (w.x*a1.y+w.y*a2.y+w.z*a3.y+w.w*a4.y)/W;
}
```

Listing 2. Implementation of the edge-directed filter. The luminance is assumed to be stored at the x component of the vectors and the chrominance (Co or Cg) at y.

(since this is the exact purpose of the RGB to YCoCg transform), a very strong correlation persists between the luminance and chrominance edges. For example, at the edge of a surface, where a chrominance transition will naturally occur, the luminance will probably change too. Although it is easy to construct an artificial case where this is not true, we have found that this assumption works very well in typical scenes. Another option would be to perform the edge-detection based on the depth values, but this would completely ignore the chrominance transitions created by the textures of the scene.

The reconstruction should be robust enough to handle the most challenging cases, like high-frequency content and strong-chrominance transitions, without introducing any visible artifacts. In Figure 3, we observe that the edge-directed filter handles these cases without any artifacts. The *peak signal to noise ratio* (PSNR) of the reconstructed frame buffer when using this filter is higher than 42dB, even for noisy content, which is very satisfactory. We also measure the perceptual difference of the resulting frame buffers, using the metric described in [Yee 2004]. The reported number is the percentage of pixels that differ from the original uncompressed frame buffer, as measured using the *pdiff* utility. These measurements and the visual inspection of the results indicate that our method provides an image quality very close to the original uncompressed frame buffers. In Figure 4, we provide some additional examples demonstrating the edge-directed filter on thin features. For the tests

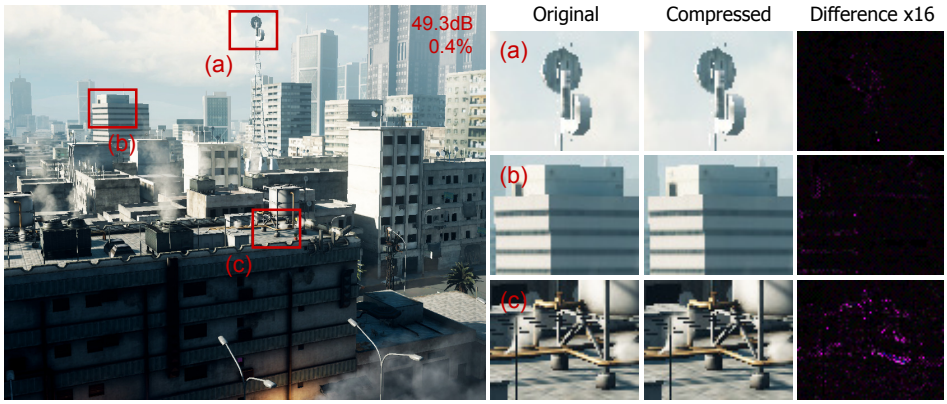

Figure 4. The edge-directed filter on thin and high-contrast features. As expected, the error is higher on extremely thin features, but still no artifacts are visible in the final frame buffer. Please note that for visualization purposes, the absolute error is magnified 16 times.

in Figures 3 and 4, we start with the uncompressed color buffers in sRGB color space (8-bit per channel), convert them to two channels by dropping the appropriate chrominance values, and finally perform the reconstruction using our filters. This process is equivalent to directly rasterizing the buffers in our compact format when no post-process blurring effects have been applied to the input frame buffer.

As can be seen in the demo application of this paper, the edge-directed filter does not produce any visible artifacts, or quality degradation, when animation is involved. To further investigate the applicability of our method in real-world scenarios, like fast action scenes, we have applied our compression scheme to a pre-recorded video, capturing the action of a modern video game. Our conclusion is that our technique can handle fast animation sequences without any problem. The interested reader is strongly encouraged to inspect the full-resolution frame buffers and watch the video on the website for this article.

5. An Investigation of Alternative Methods

In Figure 3, we investigate the quality of some alternative methods and formats that provide the same memory footprint as our technique. We recommend frequently checking the results in this figure while reading through this section.

The first alternative method simply renders the image using a frame buffer with 33% fewer pixels. This comparison is very important and educational, since reducing the resolution is the simplest and most commonly used method to save storage. Naturally, compared to this method, our scheme preserves high-frequency content more accurately. A second alternative method renders the scene using the traditional 16-bit $R_5G_6B_5$ format. Dithering is used to hide the banding artifacts caused by the reduced bit depth. This format was very popular in the early days of desktop 3D graphics accelerators. As expected, this method exhibits several color banding or noise artifacts from the dithering. In contrast, the color reproduction of our method is much more accurate, and the visual quality is very close to the original uncompressed frame buffer.

Another format we investigated encodes the fragments in the YCoCg color space and reduces the bit depth of the chrominance, thus creating a $Y_8Co_4Cg_4$ format. Again, dithering is used to hide the visible banding artifacts. Our format, as described in Section 3, trades off the spatial resolution of the chrominance, while this format trades off the bit-depth of each chrominance sample.

Although this encoding initially sounded promising, our experiments indicate that it leads to significant errors in the reproduction of colors.

We have also experimented with a much more aggressive compression format that encodes a full-color image in a single channel of data, using the Bayer mosaic pattern [Bayer 1976]. This pattern is used in most single-chip digital image sensors found in digital cameras and allows a single array of monochromatic photo receptors to capture color images by arranging a mosaic of color filters in front of them. Since this format works remarkably well for the encoding of photographs, it could have been a viable option for frame-buffer compression as well. However, this assumption proved to be wrong, because the images created by real-time rendering exhibit much higher frequencies than the ones captured by a real-world lens systems, leading to increased chrominance noise, as shown in the foliage close-up of Figure 3. Based on the above observations, we concluded that this aggressive encoding mode is not robust enough for general use. For our experiments with this format, we used the MHC reconstruction filter [Malvar et al. 2004], because it can be efficiently adapted to GPUs [McGuire 2008]. To validate the results, we also experimented with the filters in the GMIC software, but we observed similar artifacts.

Finally, we investigated an extension of the Bayer format that uses two channels. The green channel is stored in every pixel, while the red and blue channels are interleaved in a checkerboard pattern. This is essentially an adaptation of the format described in Section 3 to use the RGB color space. With this format, our edge-directed filter performs the reconstruction guided by the edges of the green color channel. In our experiments, we found that this format outperforms all the other alternatives we have discussed in this section, but the image quality is still up to 10dB lower in PSNR compared to our compact YCoCg encoding. This enormous PSNR difference is essentially the coding gain from the usage of the YCoCg color space in our method.

6. Antialiasing

The chrominance interleaving scheme described in the previous sections can be trivially adapted to hardware multisample antialiasing (MSAA). Each pixel of a multisample buffer stores multiple samples of color and depth/stencil information, each sample corresponding to a stochastic sampling position inside the footprint of the pixel [Cook 1986]. When MSAA is used with the shader of Listing 1, each pixel will store either multiple pairs of YCo data or multiple

pairs of YCg data, but never a mixture of both. This compact multisample buffer can be resolved as usual before applying the demosaicing filter of Section 4. However, the reconstruction filter should not be wider than one pixel, to avoid incorrectly mixing the Co and Cg samples. Therefore, a custom resolve pass is required if the built-in hardware resolve uses wider filters. This is hardly objectionable, since wider filters are rarely used in real-time graphics, and, if necessary, they can be applied on the luminance channel, which is perceptually the most important with respect to spatial detail.

7. Blending

Since the RGB to YCoCg transform is linear, blending and filtering can be performed directly in the YCoCg color space. This is particularly true when the frame buffer encodes radiometric values in linear color space, something that requires more than eight bits of precision in order to avoid visible banding artifacts. However, for performance reasons, real-time rendering is often performed using eight-bit precision with gamma-corrected (sRGB) values. In this case, it is worth discussing some implementation details.

First, direct blending of non-linear values is incorrect. However, it was done in many real-time applications, until sRGB buffers started to be explicitly supported by the hardware (EXT_framebuffer_sRGB extension). Such a buffer will convert the gamma-corrected values in linear space, perform the blending, and convert the results back to gamma-correct sRGB space, in order to efficiently store them in eight bits per channel. However, this non-linear operation should not be performed on YCoCg values; thus, our method cannot take advantage of hardware sRGB buffers.

Furthermore, eight-bit render targets cannot encode negative values. The RGB to YCoCg transform produces chrominance values in the $[-0.5, 0.5]$ range; thus, we have to add a bias of 0.5 in order to map them in the $[0, 1]$ range. This bias must be subtracted when reading the chrominance from the buffer. The bias will remain constant when *alpha blending* is used to render transparent objects, but with other blending modes, like *additive blending*, it will be accumulated, often leading to excessive clamping artifacts. Furthermore, the bias we have to subtract in this case is $0.5N$, where N is the number of accumulated fragments, a number which is not always known or easy to compute.

For these reasons, when blending is necessary in eight-bit render targets, we recommend the usage of the compact format in the RGB color space. An-

other option is to perform the blending operation inside the shader, in linear color space and in the correct $[-0.5, 0.5]$ range, on platforms that support it (NV_texture_barrier on Nvidia/ATI, APPLE_shader_framebuffer_fetch on iPhone/iOS6), but this solution is limited to specific hardware and use cases. These limitations only concern eight-bit render targets, but high-quality rendering typically requires higher precision floating-point formats, which are trivially handled by our method.

8. Performance

The measurements in this section were performed on a Nvidia GTX460 (768 MB RAM, 196-bits memory bus). In the first experiment, we measure the pixel fill rate, memory bandwidth, pixel storage, and reconstruction speed when rendering to a compressed render target at various bit depths. Since rasterization without a z-buffer is rarely used, the measurements shown in Table 1 include the bandwidth and storage for reading and writing to a 32-bit z-buffer. In our tests, the compressed frame buffer uses a two-channel frame-buffer format (GL_RG), while the uncompressed one uses a comparable four-channel

		Uncompressed			Compressed			
		8 bit	16 bit	32 bit	8 bit	16 bit	32 bit	
No Blending	F:	8.28	4.61	2.42	8.18	8.07	4.56	Gpixels/s
			Fill-Rate Increase:		0.99x	1.75x	1.88x	
	B:	96	128	192	80	96	128	bits/pixel
			Bandwidth Reduction:		0.83x	0.75x	0.66x	
Blending	F:	8.28	4.56	1.1	8.18	4.52	1.99	Gpixels/s
			Fill-Rate Increase:		0.99x	0.99x	1.8x	
	B:	128	192	320	96	128	192	bits/pixel
			Bandwidth Reduction:		0.75x	0.66x	0.6x	
Pixel Size		64	96	160	48	64	96	bits
			Storage Reduction:		0.75x	0.66x	0.6x	
Resolve Pass		0.55	0.75	1.0	0.56	0.56	0.78	millisec

Table 1. Comprehensive measurements of the pixel fill rate (F), memory I/O for each new fragment (B), the size of each pixel in the frame buffer (including 32-bit depth), and the resolve speed when rendering to a 720p render target with 8-, 16-, and 32-bit precision.

format (GL_RGBA), with the same precision on each channel as the uncompressed one. Using a three-channel format with the bit depths that we used in this experiment (8, 16, and 32 bits) would not be possible, due to memory alignment restrictions.

The benchmark application in the first experiment is designed to tax the fill rate of the GPU, by rendering many large visible polygons, in order to measure the improvement in this area. First, we observe that the GPU fill rate is directly proportional to the size of each pixel. The more data the GPU rasterizer has to write for each pixel, the fewer pixels per second it can fill. By reducing the size of each pixel, our method achieves an impressive 75–88% increase in the fill rate when rendering to 16- and 32-bit floating-point formats. Of course, we should mention that applications that are limited by geometry or ALU throughput will not see such an increase in the performance. In the 8-bit case, we did not measure any fill-rate increase, indicating that the 8-bit 2-channel format (GL_RG8) is handled internally as a four-channel format (GL_RGBA8). Furthermore, we did not measure any increase in the fill rate when blending is enabled on 8-bit and 16-bit render buffers, indicating some limitation in the flexibility of the ROP units in this specific GPU architecture. In all cases, the application will use up to 40% less memory for the frame buffer. This memory savings can be used to store more textures and meshes. As an example, an uncompressed 1080p render target with 8xMSAA requires 189 MB of storage at 16-bit half-float precision, while with our method it requires only 126 MB. Both numbers include the z-buffer storage.

It is also interesting to examine the bandwidth required to rasterize a new fragment in the frame buffer. For a visible fragment, the GPU has to read the old 32-bit depth value from the z-buffer in order to perform the depth test, and then it has to write back the new depth and color information. When blending is enabled, the old color should also be fetched. The total number of bits for each case is shown in Table 1. Based on this analysis, we can calculate that, for a 16-bit render target, our technique reduces the bandwidth consumption by 25% without blending, and by 33% when blending is enabled. We should also mention that during a z-fail, only the old z-value has to be fetched. This case is not affected by our technique, since no actual rasterization is taking place.

In this experiment, we also measure the time it takes to resolve (blit) a compressed 720p render buffer to the GPU back buffer. This operation is performed by rendering a full-screen quad that uses the render buffer as a texture. In this pass, applications can also perform tone mapping and other post-

processing operations. We observe that blitting a compressed render buffer is faster than blitting an uncompressed one, as shown in Table 1, since less data need to be fetched from memory. In particular, when using a half-float 720p frame buffer, the resolving process is 0.19 ms (25%) faster. The small increase in the ALU instructions, needed to decode the data, is counterbalanced by the reduction in memory bandwidth. Although our OpenGL implementation has to perform four additional unfiltered texture fetches per pixel, to feed the edge-directed reconstruction filter of Listing 2, most of these fetches will come from the texture cache, which is very efficient in most GPUs. A GPGPU implementation can completely avoid the redundant fetches, by leveraging the local shared memory of the ALUs, but we chose to focus on the OpenGL implementation, since GPGPU capabilities are not available on all platforms. In this experiment, we used the edge-directed filter to de-multiplex the chrominance data, but the less complex filters performed the same, indicating that the blit operation is bandwidth-limited, and not ALU-limited. Of course, these measurements could also indicate that the ALUs are rather underutilized by our test application. Thus, we encourage developers to measure the actual performance in their particular application.

In our second experiment, we integrated our method into a deferred-lighting pipeline, a practical algorithm used by many modern game engines. This pipeline involves the accumulation of the diffuse and specular light in two buffers. Readers unfamiliar with deferred rendering are referred to [Akenine-Möller et al. 2008]. Many implementations of deferred lighting reduce the memory footprint of the algorithm by dropping the chrominance information of the specular buffer, in order to store both buffers in one render target. This

Reference - six channels Drop chroma - four channels Ours - four channels

Figure 5. Integration of our algorithm in a deferred-lighting pipeline. Left: Accurate diffuse and specular accumulation using six channels (two render targets). Middle: Fast but inaccurate accumulation using one render target, by dropping the specular chrominance. Right: Fast and accurate accumulation, using our compact format to store the diffuse and specular buffers in one render target.

approximation can lead to incorrect specular highlights, as shown in Figure 5 (middle). In this example, a red and green light source are shining onto a surface from different directions. The diffuse term will be yellow in the middle, but the specular term should have two separate highlights, one red and one green. This detail in the lighting is lost when dropping the chrominance information. On the other hand, our method can be used to accumulate both the diffuse and specular lighting in one render target, without compromising the accuracy of the specular highlights or the rendering speed, as shown in Figure 5.

9. Discussion and Limitations

In this article, we presented a lossy frame-buffer compression format that performs chroma subsampling by storing the chrominance of the rasterized fragments in a checkerboard pattern. This simple idea allows the rasterization of a color image using only two color channels, saving both storage space and memory bandwidth and at the same time increasing the rasterizer fill rate.

The solution is simple to implement and can work in commodity hardware, like game consoles. In particular, the memory architecture of the Xbox 360 game console provides a good example of the importance of our method in practice. Xbox 360 provides 10 MB of extremely fast embedded memory (edram) for the storage of the frame buffer. Every buffer used for rendering, including the intermediate buffers in deferred pipelines and the z-buffer, should fit in this space. To this end, our technique can be valuable in order to fit more data in this fast memory. Bandwidth savings are also extremely important in mobile platforms, where memory accesses will drain the battery.

Our technique can also provide some small additional benefits during the fragment shading, where the fragments can be converted to the two-channel interleaved format early in the shader code and then any further processing can be performed only on two channels, instead of three, in the YCoCg color space. Measuring this benefit is beyond the scope of this paper, since the actual gain depends on the shading algorithms involved and the architecture of the underlying GPU, and, in particular, the mix of scalar and SIMD units.

A rather obvious limitation of our method is that it can only be used to store intermediate results and not the final device frame buffer, because the hardware is not aware of our custom format. However, this does not limit the usefulness of our method, since most modern real-time rendering pipelines use many intermediate render buffers before writing to the back buffer. An-

other limitation of our method is that hardware texture filtering cannot be used to fetch data from the compressed frame buffer. To sidestep this limitation, developers should use the reconstruction filters of Section 4 to de-multiplex the packed channels into one luminance and one chrominance texture. Since the chrominance texture will have a lower resolution, the application will enjoy a reduction in bandwidth usage. It is worth noting that none of the above limitations would exist if the hardware incorporated support for our custom packed format. Finally, in case it is not clear, we should note that our method should be used only on color buffers, since chroma subsampling makes little sense in any other kind of data.

Acknowledgements

We would like to thank Stephen Hill (Ubisoft Montreal) for his very insightful comments on the technique. We would also like to thank Charles Poynton for his insights on color spaces. The Epic Citadel scene was captured with the publicly available Unreal Development Kit (UDK). *Battlefield 3* images are courtesy of EA Digital Illusions Creative Entertainment; *Crysis 2* images are courtesy of Crytek GmbH.

References

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A K Peters, Ltd., Natick, MA, USA. 32

BAYER, B. 1976. *Color imaging array*. United States Patent 3971065. 28

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Trans. Graph. 5* (January), 51–72. 28

MALVAR, H., AND SULLIVAN, G. 2003. *YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range*. Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, Document No. JVTI014r3. 21

MALVAR, H., HE, L.-W., AND CUTLER, R. 2004. High-quality linear interpolation for demosaicing of Bayer-patterned color images. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings (ICASSP '04), IEEE International Conference*, vol. 3, iii – 485–8. 28

MCGUIRE, M. 2008. Efficient, high-quality Bayer demosaic filtering on GPUs. *Journal of Graphics Tools 13*, 4, 1–16. 28

OWENS, J. 2005. *Streaming Architectures and Technology Trends*. Addison Wesley, Reading, MA, USA, 457–470. 20

SALVI, M., AND VIDIMČE, K. 2012. Surface based anti-aliasing. In *ACM SIGGRAPH Symposium on Interactive 3D Rendering and Games*, ACM, New York, NY, USA. 20

YEE, H. 2004. A perceptual metric for production testing. *journal of graphics, gpu, and game tools 9*, 4, 33–40. 26

Index of Supplemental Materials

In the supplemental materials for this article, the interested reader can find a live WebGL demonstration of the method, as well as source code and a video that demonstrate the technique when encoding data from a modern game.

Author Contact Information

Pavlos Mavridis
Department of Informatics,
Athens University of Economics
& Business
76 Patission Str.
Athens, 10434 Greece
pmavridis@gmail.com
http://pmavridis.com

Georgios Papaioannou
Department of Informatics,
Athens University of Economics
& Business
76 Patission Str.
Athens, 10434 Greece
gepap@aueb.gr
http://www.aueb.gr/users/gepap

Pavlos Mavridis and Georgios Papaioannou, The Compact YCoCg Frame Buffer, *Journal of Computer Graphics Techniques (JCGT)* 1, 1, 19–35, 2012
http://jcgt.org/published/0001/01/02/

Received:	6 June 2012		
Recommended:	27 July 2012	Corresponding Editor:	Naty Hoffman
Published:	30 Sept. 2012	Editor-in-Chief:	Morgan McGuire