High Performance Graphics (2012)
C. Dachsbacher, J. Munkberg, and J. Pantaleoni (Editors)

Adaptive Scalable Texture Compression

J. Nystad[1], A. Lassen[1], A. Pomianowski[2], S. Ellis[1] and T. Olson[1]

[1]ARM [2]AMD

Abstract

We describe a fixed-rate, lossy texture compression system that is designed to offer an unusual degree of flexibility and to support a very wide range of use cases, while providing better image quality than most formats in common use today. The system supports both 2D and 3D textures, at both standard and high dynamic range, at bit rates ranging from eight bits per pixel down to less than one bit per pixel in very fine steps. At any bit rate, texels can have from one to four color components. The system's flexibility results from a number of novel features. Color spaces and weights are represented using an encoding scheme that allows flexible allocation of bits between different types of information. The system uses bilinear interpolation to derive color space coordinates for a texel from sparse samples, and uses a procedural partition function to map texels to color spaces.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Texture

1. Introduction

Textures are a fundamental computational primitive in modern 3D graphics [AMHH08], and are heavily used in modern applications. Because of this, texture accesses often make up a large fraction of graphics accelerator memory bandwidth, which is frequently the limiting factor in system performance and power dissipation. Texture access bandwidth can be reduced by rendering from compressed textures, as introduced by Knittel et al [KSKS96], Beers et al [BAC96], and Torborg and Kajiya [TK96]. The technique has been widely adopted, and there is great interest in improved texture compression methods, especially for mobile devices [AMS03].

Textures play many different roles in graphics applications; texture data may represent surface properties (including specular and diffuse reflectance, opacity, surface orientation and height), illumination (both in the environment and at the surface), density, or other physical properties, sometimes as 3D textures. Each of these use cases has characteristic requirements for number of color channels (i.e. number of values per texel), dynamic range, and quality.

Many compressed texture formats have been developed, but few address more than a small subset of the use cases described above. Most support only one or two bit rates, and one or two choices of number of color components. Table 1 shows how a number of common formats fit into the space of format parameters. There are many gaps, and no single format covers very much of the space; support for low bit rates, low numbers of channels, and HDR is especially limited.

channels	2 bpp	4 bpp	8 bpp
1	-	RGTC	uncompressed
2	-	-	RGTC
3	PVRTC	S3TC PVRTC ETC1	 BC7 BC6H
4	PVRTC	S3TC PVRTC	DXT5 BC7

Table 1: *Common compressed texture formats grouped by bit rate and channel count.*

In this paper we describe ASTC, a new format that attempts to address the widest possible range of use cases and to provide unprecedented flexibility for applications. ASTC supports bit rates ranging from 8bpp down to less than 1bpp in very fine steps, providing a high degree of control over the speed *vs* quality tradeoff. At any bit rate, texels can have from one to four color components. High dynamic range texels are supported for one, three, or four component textures, and the format can encode 3D as well as 2D textures.

In addition to offering exceptional flexibility, ASTC provides a substantial improvement in visual quality compared to most other formats in common use. It outperforms S3TC

DOI: 10.2312/EGGH/HPG12/105-114

and PVRTC by as much as two dB (PSNR) at comparable bit rates, and is competitive with advanced formats such as BPTC. The silicon cost of an ASTC decoder is higher than that of other high-end formats such as BPTC, but is justified by its exceptional flexibility and quality.

ASTC's features are the result of several technical innovations. The most fundamental of these is a low-level coding scheme called *bounded integer sequence encoding*, which allows sequences of data values to be represented using a fractional number of bits per value. This makes it possible to allocate bits between different types of information in a very flexible way without sacrificing coding efficiency. Other novel features include the use of bilinear interpolation to derive color weights for a texel from sparse samples, and use of a procedural classification function to specify the partition index that maps a texel to a color space.

In the next section we review related work, focussing on fixed-rate, block-based compression systems intended for use in hardware graphics accelerators. We then describe ASTC and explain how it differs from other block-based compression formats. The last section of the paper presents performance results and quality comparisons against a number of well-known formats.

2. Related Work

Most texture compression formats in use today follow the block-based paradigm introduced by Delp and Mitchell [DM79] and later generalized in various ways [FNK94]. In these approaches, the image is divided into fixed-size blocks, and each block is encoded (lossily) as a fixed-length bit vector. The bit rate is given by the ratio of vector length to number of texels per block. The block-based paradigm has the advantage that any texel can be decoded using a constant number of memory references and a constant amount of computation. It is the basis of the majority of texture compression formats in use today, including S3TC [INH99], ETC1 [SAM05], and BPTC [Ope10].

In formats of this type, resolving a texture reference requires dividing the texture coordinates by the block width and height; the quotients indicate which block contains the desired value, and the remainders give its offset within the block. Block dimensions are usually chosen to be powers of two, making the division and remainder computations trivial, and blocks are typically square. Exceptions include the POOMA format [AMS03], which uses 3×2 blocks, PACKMAN [SAM04] (2×4), and PVRTC 2bpp [Fen03] (8×4).

PVRTC [Fen03] is a fixed-rate format, but is not block-based in the conventional sense. Texels are reconstructed by upsampling two low-resolution images and interpolating between them using a per-texel interpolation weight. It is less prone to block artifacts than block-based systems, but it tends to blur details, and the overlapping blocks make encoding a global optimization problem.

2.1. Block Contents

Compression systems using the block-based paradigm store three basic types of information in the bit vector encoding of each block of texels:

color space: A set of colors from which the colors assigned to texels in the block may be chosen.

color specifiers: A set of data values used to associate individual texels with points in the block's color space.

control information: Additional bits used to select between multiple color spaces, specifier types, or association methods, or to supply any additional parameters needed.

In early work [DM79, CDF*86], the color space consists of two grey values or colors. Color specifiers consist of one bit per texel, which is used to select one of the two colors. The simplicity of the scheme and the lack of control information allow the bit rate to be very low (2bpp), but make it impossible to represent smooth gradients across a block. The S3TC system of Iourcha et al [INH99] extends this idea by specifying two RGB colors A and B explicitly, and inferring two others equally spaced along the line segment $\overline{AB}$. The color space for this format consists of a discrete set of four colors, and color association is done by using a two-bit-per-texel index to select one of the four.

The PACKMAN format of Ström and Akenine-Möller [SAM04] introduces a color space consisting of a single RGB base color modified by the addition of four different luminance offsets. The resulting colors are associated with specific texels using a two-bit-per-texel index. The four offsets are chosen from a table of four-vectors, indexed by four bits of per-block control information.

A number of formats have been derived from the basic PACKMAN scheme. Improved PACKMAN (iPACKMAN) [SAM05] combines two adjacent PACKMAN-style blocks into a single 4×4 block, and provides the option of using differential coding to specify the color spaces for the two sub-blocks. The sub-blocks can be either horizontally or vertically oriented, which can be seen as a form of partitioning (see below). iPACKMAN has been standardized as ETC1 [Khr05], and is widely supported on mobile devices. The most recent version of PACKMAN is ETC2 [SP07], which introduces new color space options to handle blocks that contain large chrominance variations. ETC2 provides excellent quality and represents the state of the art for RGB compression of natural image textures at 4bpp.

The most complex block-based formats in commercial use today are the BPTC [Ope10] formats, BC6H and BC7. BC7 is used to compress conventional low dynamic range (LDR) textures, while BC6H supports high dynamic range images. The formats offer a large number of modes specifying different color space representations and color assignment schemes. BPTC introduces the concept of *partitions*; each texel in a block is assigned to one of two or three parti-

tions, each of which has its own color space. Control information in the block selects a table that maps a texel position to a partition index, which in turn selects a color space. As in S3TC, color spaces consist of pairs of color endpoints, and per-texel weights are used to interpolate between endpoints in the selected color space. The per-texel weights can have two to four bits per texel depending on color mode.

2.2. The Bit Allocation Problem

One of the fundamental problems that must be solved in designing a compressed texture format is that of how to divide the available bits between the three kinds of information that the block must store: color space(s), color specifiers, and control information. Color spaces and specifiers typically consist of sequences of values in some fixed range; for ease of decoding, each value is usually stored in an integer number of bits. However, this imposes a coarse granularity on the allocation of bits between colors and specifiers. For example, in BC7 the number of bits devoted to color weights must be two, three, or four bits per weight, i.e. 32, 48, or 64. This limits the options for how many bits can be used for color space and control information.

The bit allocation problem becomes easier if it is possible to represent values using a fractional number of bits. Wennersten and Ström [WS09] describe a 2bpp format in which color specifiers take on values in the range [0..2]. Values are grouped into pairs, and three bits are used to specify one of eight combinations of values at a rate of 1.5 bits per index. In the POOMA format [AMS03], groups of three color specifiers in the range [0..2] (which can take on 27 values) are described by 32-bit codes at a rate of 1.67 bits per symbol.

3. The ASTC Compressed Texture Format

Our compressed texture format is called ASTC, for 'adaptive scalable texture compression'. It is *scalable* in the sense that a single hardware design scales across a very wide range of bit rates, and *adaptive* in the sense that most information is specified on a per-block or per-partition basis. The only global properties of an ASTC image are its dimensionality (2D or 3D) and its bit rate; this allows the encoder to adapt to local variations in statistics within a single image.

ASTC follows the standard block-based paradigm and shares many features with the formats described in the previous section. Images are partitioned into fixed-size blocks, which are encoded as vectors of 128 bits. As in S3TC, color spaces are specified by pairs of points that define line segments in color space. As in BPTC, texels are assigned to partitions associated with color spaces, and per-texel color weights interpolate between color endpoints.

3.1. Bit Rates

The most obviously unusal feature of ASTC is that it supports a very large number of block sizes and hence bit rates.

2D block size	bit rate	3D block size	bit rate
4×4	8.00 bpp	3×3×3	4.74 bpp
5×4	6.40 bpp	4×3×3	3.56 bpp
5×5	5.12 bpp	4×4×3	2.67 bpp
6×5	4.27 bpp	4×4×4	2.00 bpp
6×6	3.56 bpp	5×4×4	1.60 bpp
8×5	3.20 bpp	5×5×4	1.28 bpp
8×6	2.67 bpp	5×5×5	1.02 bpp
10×5	2.56 bpp	6×5×5	0.85 bpp
10×6	2.13 bpp	6×6×5	0.71 bpp
8×8	2.00 bpp	6×6×6	0.59 bpp
10×8	1.60 bpp		
10×10	1.28 bpp		
12×10	1.07 bpp		
12×12	0.89 bpp		

Table 2: *ASTC supported block sizes and bit rates. All block sizes are compressed to vectors of 128 bits, resulting in the bit rates shown.*

Blocks are not required to be square, and block dimensions are not required to be powers of two. Table 2 shows the set of supported block dimensions and the associated bit rates.

The hardware cost of supporting this large number of block sizes is dominated by the cost of the quotient and remainder operations needed to map a texel address to a block and an offset within the block. Since most of our block sizes are not powers of two, these computations are nontrivial. However, all of the block sizes can be expressed as products of three, five, and powers of two. This fact can be exploited to simplify the address computation logic.

ASTC's large blocks may cause it to produce smaller memory bandwidth savings than bit rate alone would suggest, since at silhouette edges the texture cache will fetch more unused texels than a format with smaller block sizes. However, such overfetch effects are likely to be dominated by cache line size rather than by texture block size, so we expect any effect to be subtle.

3.2. Bounded Integer Sequence Encoding

A key feature of ASTC is that both color space and color specifier values are represented using a fractional number of bits. This gives the encoder great flexibility in deciding how accurately to encode each type of information. The method is called *bounded integer sequence encoding*, because it operates on sequences of integers lying in a restricted range.

Bounded integer sequence encoding (BISE) addresses the following abstract problem: Given sequences of equiprobable values in the range 0 to $N-1$, find an encoding that allows the ith value to be extracted in constant time with minimal hardware, allows the same hardware to be used with many different ranges of values, and is storage-efficient. A simple solution that meets the first two requirements is to encode sequences as packed bit strings of binary numbers. This

scheme is storage-efficient when N is a power of two, since the number of bits of information in a single value ($log_2 N$) is exactly the number of bits used by the encoding; that occurs when values are represented by an integer number of bits.

BISE extends the above solution by adding almost perfectly efficient storage when N is 3×2^n or 5×2^n. For $N > 4$, this has the effect of introducing two new optimal value ranges in between each successive pair of powers of two. In the former case, the two most significant bits of each value specify a *trit* or trinary digit, which can take on values 0, 1, and 2. Similarly, in the latter case, the three most significant bits of a value are considered a *quint* or base-5 digit. For example, if $N = 12$, values are represented using one trit and two bits: $X = t2^2 + b_1 2^1 + b_0 2^0$.

To encode a sequence of values that is represented using trits, we partition the sequence into groups of five values, padding the last group with zeros if necessary. Conceptually, each group can be represented as the bit string $t_4B_4t_3B_3t_2B_2t_1B_1t_0B_0$, where the t_i are two-bit representations of trits, and the B_i are the remaining bits of each value. Since five trits can take on $3^5 = 243$ values, they can be stored in eight bits. Similarly, sequences based on quints can be represented as $q_2B_2q_1B_1q_0B_0$, and since three quints can take on 125 values, they can be stored in seven bits. The compressed groups remain interleaved with the bit sequences; if the trit string is compressed to an eight-bit value T, then the compressed group of five values is $T_{[7]}B_4T_{[6:5]}B_3T_{[4]}B_2T_{[3:2]}B_1T_{[1:0]}$. The analogous interleaving for quint groups is $Q_{[6:5]}B_2Q_{[4:3]}B_1Q_{[2:0]}B_0$.

The encoding and interleaving scheme for the trit and quint groups has the special property that it preserves trailing zeros in the compressed trit and quint strings. In the trit case, for example, if the last group of five values was padded with two zeros (so that t_4B_4 and t_3B_3 are known to be zero), then in the compressed string it is also the case that $T_{[7]}B_4$ and $T_{[6:5]}B_3$ are zero; and since the values are known, they do not have to be stored. At decode time, any references to bits beyond the end of the stored bitstring can be satisfied with zeros, and the decoding process will work correctly. Thus the encoding scheme remains efficient even if the length of the value sequence is not a multiple of three or five.

The encoding scheme described above represents n trits with $\lceil \frac{8n}{5} \rceil$ bits, and n quints with $\lceil \frac{7n}{3} \rceil$ bits. As sequence length increases, the encoding rates approach asymptotic limits of 1.6 bits per trit and 2.333 bits per quint, which is very close to the information-theoretic bounds of $log_2(3) \approx 1.585$ bits per trit and $log_2(5) \approx 2.322$ bits per quint.

3.3. Partitions

Every texel in an ASTC block is assigned to one of up to four partitions, each of which has its own color space. Which partition a texel belongs to is determined by a function that maps texel position to an integer in the range zero to three. This use of partition functions is similar to BPTC, but where BPTC's function is stored as tables, ASTC's is computed from the partition count and the partition ID. Pseudocode for the classification function is available as supplemental material.

The function takes as input the ten-bit partition ID, the number of desired partitions (two to four), and the coordinates of the texel within the block. It appends the partition count minus one to the partition ID, and uses the result to seed a procedural pseudorandom number generator. The 32-bit result is split into fields and used to produce four-bit coefficients for four planar functions of position in the block. The functions are evaluated at the desired texel location, and the resulting values are masked to extract the six least significant bits. This converts the planes into sawtooth functions of random frequency and orientation. The masked values are compared, and the index of the function which produced the largest value is returned as the partition index for the texel. The resulting partition patterns tend to consist of coherent regions with piecewise linear boundaries.

The partition function is designed to have very low silicon cost. It uses no tables, and all operations are performed on quantities of 32 bits or less. All multiplications are performed on four-bit quantities and so are inexpensive. All shifts are by constant values, so they require no logic. The result is that the silicon area is a small constant, rather than being proportional to the number of partition patterns as in BPTC. This makes it possible to implement a large set of partition patterns in a relatively small number of gates.

A disadvantage of our procedural partition function is that about 50% of the generated patterns are useless for one reason or another: they are redundant, or return constants, or are too fragmented to be useful. Thus, about one bit of the ten-bit partition ID is wasted. Adopting BPTC's table-based approach for ASTC would allow one bit of partition ID to be used for other purposes. However, we believe that the storage cost of the tables would be prohibitive, due to ASTC's much larger block size and the need to support both 2D and 3D partition functions.

3.4. Color Spaces

ASTC follows BPTC, S3TC, and PVRTC in using color spaces consisting of pairs of color endpoints. Texel colors are determined by a per-texel weight that interpolates between the endpoints, as will be described in the next section.

ASTC provides sixteen modes for specifying color endpoint pairs; ten describe LDR color spaces, and the remaining six describe HDR spaces. Each is designed so that all values are expressed as integers in a common range, so that they can be encoded efficiently using BISE. For LDR color modes, color endpoints are represented by fixed-point values in the range [0..1], dequantized to eight bits after extraction from the BISE encoding. For the HDR modes, values

are dequantized to eight bits and then processed in mode-dependent ways to produce 12-bit unsigned floating point values with five bits of exponent and seven bits of mantissa.

The various modes make use of a number of basic encoding methods, which we will explain here in order to simplify the description of the modes:

independent The independent methods specify two k-bit values directly.

base+offset These methods are variants of iPACKMAN's differential encoding scheme [SAM05]. They specify two values as a $(k+1)$-bit base and a $(k-1)$-bit offset; the base provides one value, and the offset added to the base provides the other. This improves resolution when two values are close together. One bit of the base value is stored in the offset, converting them into k-bit integers.

base+scale The base+scale methods encode two RGB values using four numbers; the four-tuple (R,G,B,s) is interpreted as specifying the colors (R,G,B) and (sR,sG,sB). This mode is useful when most of the color variation in a partition is along the luminance axis.

The color modes themselves are classified by number of values they specify: two, four, six, or eight. There are four modes of each class.

two-value modes These modes represent luminance-only (single channel) color spaces. Endpoints are specified either directly or using the base+offset method. Two modes represent LDR values. One of the two HDR modes simply left-shifts the dequantized integers to form 12-bit values, while the other uses a base+offset method, transferring several bits from the second value to increase the resolution of the first.

four-value modes Four-value modes represent two and three-channel color spaces. Two modes specify LDR luminance and alpha as independent values or as base+offset. Two other modes specify two RGB endpoints by the base+scale method, one as LDR and the other as HDR. In the HDR mode, four bits taken from the values specify a sub-mode which controls the distribution of six more of the bits to the values (R,G,B,s), allowing the encoder to represent different components at different resolutions.

six-value modes Six-value modes represent three and four-channel color spaces. Two LDR modes specify pairs of RGB colors using independent values or base+offset. A third LDR mode specifies two RGB colors using base+scale, and interprets the last two values as independent alphas. The last mode specifies two HDR RGB endpoints. Five bits taken from the values define a sub-mode that is used to shift bits between components.

eight-value modes Eight-value modes specify four-channel RGBA color spaces. Two modes specify LDR colors using independent values or base+offset. The other two specify HDR RGB endpoints as in the six-value mode, and add two alpha values. One mode provides normalized (LDR) alpha, supporting the common case where an HDR image has alpha values in the range $[0,1]$. The other mode provides HDR alpha, with a sub-mode bit to select either independent or base+offset conversion.

It is worth reiterating that color spaces are specified on per-partition basis. Choosing a mode for one block places no resriction on modes for other blocks. The encoder is free to (and often does) encode one partition as LDR, while another in the same block is encoded as HDR. The color space encoding does impose one restriction: color modes within a single block can differ by at most two in number of values, e.g., two-value modes can be combined with four-value modes, but not with six- or eight-value modes. This reduces the number of bits needed to specify the color modes, at the cost of disallowing some infrequently used mode combinations.

3.5. Color Weights

As in several other block-based formats, texel colors in ASTC are specified by per-texel values that are used to interpolate between color endpoints. We refer to these values as *color weights*. Unlike most similar formats, ASTC allows color weights to be specified at lower resolution than the block dimensions. Consider the 2D case with a block size of $M\times N$. Color weights are specified as an array of size $p\times q$, where $p \leq M$ and $q \leq N$. To derive a weight for a given texel, its position within the block is scaled to the dimensions of the weight array, and the weight at that position is obtained by bilinear interpolation. The 3D case is similar, except that we use simplex interpolation (see e.g. Gustavson [Gus05]) rather than trilinear. Simplex interpolation in 3D requires the same number of color weight values (four) and arithmetic operations as bilinear interpolation, while trilinear interpolation would require twice as many of each.

This method of specifying color weights makes it possible to trade off spatial resolution of the weight array against precision of the weights and/or the color space. For example, very smooth gradients can be obtained even in large blocks by specifying the weights as a 2×2 grid and interpolating them across the block.

Once per-texel color weights have been computed, they are dequantized to the logical range [0..1] and used to interpolate between color endpoints. For LDR endpoints, interpolation is linear. For HDR endpoints, interpolating the 12-bit floating point values as if they were integers (as is done in BC6H) would give a piecewise linear approximation to logarithmic interpolation. ASTC also uses a piecewise linear approximation, but divides each exponent range into three linear segments in order to obtain a smoother curve. Values are interpolated with a slope of 3/4 for mantissas in the range 0..0.25, a slope of 1 for mantissas in the range 0.25..0.75, and a slope of 5/4 for those in the range 0.75..1.

3.6. Decode Process

The bit vector encoding an ASTC block is divided into seven major sections. Space does not permit a detailed description of the bit layout of each section, but we will describe the information that each contains, and explain the decode process. The sections are:

index mode The index mode is stored in the first eleven bits of the block. It specifies the dimensions of the color weight array and the range of values color weights can take on.

partition count The partition count (minus one) is stored in a two-bit field following the index mode.

partition ID If the partition count is greater than one, the partition count bits are followed by a ten-bit partition ID that, together with the partition count, specifies the partition pattern as described in section 3.3.

fixed color mode data If there is only one partition, the partition count is followed by four bits which specify which of the sixteen color modes (see section 3.4) is used for the block. If there are multiple partitions, the partition ID is followed by six bits which specify how many color values are needed to describe the color spaces for the partitions, and how many extra color mode bits are needed to fully specify the color modes for each partition.

color endpoint values The fixed color mode data is followed by a variable-length bit string containing the BISE encoding of the values needed to specify the color endpoints for each color space.

extra color mode bits Any extra bits needed to describe the color modes are stored just before the color weights.

color weights The remainder of the block contains the BISE encoding of the color weight array in raster order. The encoding is stored in bit-reversed order, beginning at the end of the block and extending toward the other data items.

Two things are noteworthy about the block layout. First, the encoding of the color weights begins in a fixed location, and the encoding of the color endpoint values begins at one of only two locations. This makes it easier to route the bits that encode the desired values to the BISE decoding logic. Second, there is no explicit encoding of the range of values that the color endpoint values can take on; only the number of values is specified. The range of the values is inferred from the amount of space available to hold them. The size of all other data in the block can be calculated; subtracting it from 128 gives the number of bits available to store color endpoints. The color endpoint values are always stored using the largest range of values that will fit into those bits.

The process of decoding a texel from a 2D block is as follows: First, the index mode is unpacked to determine the size of the color weight array and the range of values weights can take on. This is sufficient to determine the length of the color weight BISE bit string, and hence the position of the extra color mode data, if there is any. The texel coordinates are rescaled to the dimensions of the color weight array, and processed to determine which color weights are needed to perform bilinear interpolation. The appropriate sections of the color weight bit string are decoded as described in 3.2, and the color weight for the texel is computed.

Second, the partition count is examined. If there is more than one partition, the partition ID, partition count, and texel coordinates are sent to the partition function generator described in section 3.3, and the partition index of the texel is determined. The partition count also determines the location and format of the fixed color mode data.

Third, the fixed color mode data are interpreted to determine how many color values are needed to specify the color endpoints, how many extra color mode bits are present, and how many bits are available to store the color endpoint values. From this, the range of values each color endpoint value can take on is computed.

Fourth, the texel's partition index is used to determine which color endpoint pair is used for the texel, and the color mode for that pair is determined. If there is only one partition, the mode is specified directly. Otherwise, the fixed and extra color mode bits are used to recover the mode and the number of values used to specify the color endpoints for each partition. This information is used to locate the appropriate values in the color endpoint bit string, the values are extracted, and the endpoints are recovered.

Finally, the color endpoint values are interpolated using the color weight as specified by the color mode for the partition, and the resulting color is returned.

4. Implementation

ASTC is intended to be a practical replacement for all of the formats shown in Table 1. In order to verify that it is suitable for mobile device silicon, we have implemented a decoder in synthesizable RTL, and a software codec that is used to encode images and to validate the hardware decoder.

The hardware decoder design is based on that of our BPTC decoder. ASTC and BPTC have the same encoded block size (128 bits), and both support 4×4 blocks, so they impose the same requirements on cache architecture and can share some addressing logic. Currently the ASTC decoder is approximately twice the area of the BPTC decoder, so it is large by current standards. However, we feel that its large number of bit rates, flexible number of color components, support for low-bit-rate HDR and 3D, and exceptional quality at low bit rates more than justifies the additional cost.

Building a fast encoder for ASTC is challenging due to the large search space. Our encoder is structured as a branch-and-bound search over the full set of partition functions, color weight array sizes, and color modes. For a given partition and color mode, we identify a dominant axis and perform a limited search to select endpoints on that axis. This

search is done for each possible number of components, and the ones yielding the lowest MSE for the block are selected. This algorithm produces very good results, but can take hours for large images, so it is too slow for practical use in a content development pipeline.

To address the speed problem, the encoder offers four 'fast' modes that trade quality for speed to varying degrees. These modes make use of a number of heuristics:

- The initial lower bound for the branch-and-bound search is set based on bit rate and speed mode. When an encoding is found at achieves the bound, further search is skipped.
- The encoder searches partition counts in increasing order. If the best two-partition encoding does not improve significantly on the one-partition encoding, the faster modes do not consider three- and four-partition encodings, since they are unlikely to provide further benefit.
- In the fastest modes, the encoder tests only the most commonly used index modes.
- The encoder orders the search over partition patterns so that 'good' partitions are visited early. It does this by performing a few cycles of k-means clustering (for $k \in \{2,3,4\}$) on the block color values, labeling each texel with its cluster number, and computing a match score between this labelling and each of the partition patterns. Patterns are then searched in order of descending match score.

We ran the encoder in single-threaded mode on a 2.97 GHz Xeon server. On the 24 images of the 'Kodak' set (size 512×768), average running times range from 300 seconds in exhaustive mode down to 0.8 seconds in the fastest mode, with a loss of about 2.5 dB in quality. The intermediate mode averages 12 seconds with very little quality loss. The implementation supports multithreading and gets good speedup at up to eight threads.

5. Evaluation

Space does not permit a detailed evaluation of the full range of use cases that ASTC supports, so we will focus on 2D RGB textures at low and high dynamic range. We compare ASTC to the leading alternatives at each bit rate:

- At 2bpp we compare ASTC to PVRTC 2bpp, compressed using Imagination Technologies' PVRTexTool, SDK build 2.10@905358.
- At 4bpp, we compare ASTC 6x6 (at 3.56 bpp) to PVRTC 4bpp, S3TC (compressed with AMD's The Compressonator 1.50.1731), and ETC2 (using a compressor provided by Jacob Ström).
- At 8bpp, we compare ASTC to BC7 for LDR and to BC6H for HDR, using the 'zohc' and 'avpclc' codecs distributed by NVIDIA [NVI].

In all cases, we used the highest quality settings provided, with colors equally weighted. For LDR images, we follow

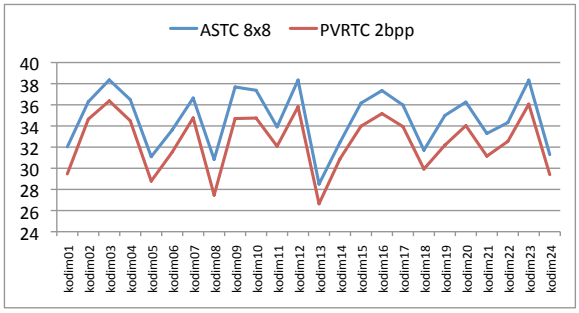


Figure 1: *LDR comparison (dB PSNR) at 2 bpp.*

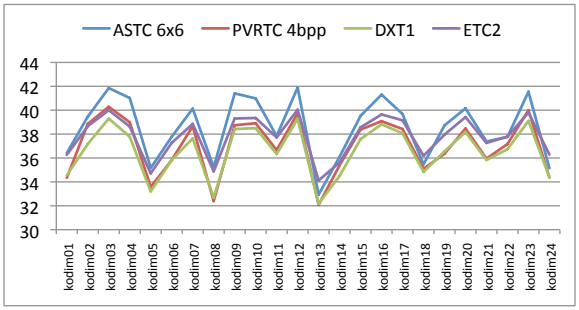


Figure 2: *LDR comparision (dB PSNR) at 3.56/4 bpp.*

standard industry practice and measure quality in terms of peak signal-to-noise ratio (PSNR), using the formula given by Ström and Petterson [SP07]. For HDR images, we use the mPSNR metric of Munkberg et al [MCHAM06].

5.1. Low Dynamic Range Results

Figures 1-3 give PSNR scores in dB across the 24 images in the well-known 'Kodak' test set [Kod]. At 2bpp, ASTC outperforms PVRTC by 2.2 dB on average. At 3.56 bpp, it improves on PVRTC and S3TC 4bpp modes by 1.5 and 1.9 dB respectively despite an 11% bit rate disadvantage. It is generally believed (see e.g. Ström and Akhenine-Möller [SAM05]) that a PSNR difference 0.25 dB is visible to most observers, so these are very significant differences. ASTC also outperforms ETC2 by 0.7 dB at the same bit rate disadvantage. As far as we know, no other fixed-rate format offers comparable quality on natural images at 4bpp or below.

The BC7 format consistently outperforms ASTC's 8bpp mode by a small margin (an average of 0.5 dB on this dataset). At 8bpp both ASTC and BC7 compressed images have PSNR quality around 45 dB, and we find the quality difference very difficult to detect visually.

Figure 4 shows example images from the 2bpp test. We also include results for the ASTC 12x10 format at 1.07 bpp. The top row image is very well suited to ASTC's single-channel luminance and base+scalar color modes. PVRTC

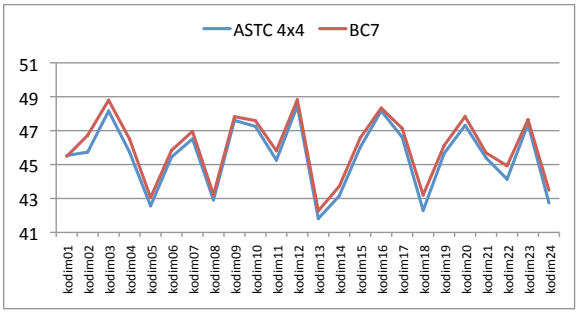


Figure 3: *LDR comparison (dB PSNR) at 8 bpp.*

Image	ASTC	BC6H	difference
Starfield	51.3	50.2	1.1
Bonita	47.7	47.1	0.6
Desk	40.5	42.6	-2.1
Memorial	43.4	44.4	-1.0
Cathedral	39.7	40.8	-1.1
BigFogMap	51.1	51.2	-0.1
Belgium	46.6	46.5	0.1
AtriumNight	52.5	51.2	1.3
MtTamWest	42.5	42.3	0.2

Table 3: *HDR comparison at 8bpp (dB mPSNR).*

blurs away detail at the right side of the image, and is unable to represent the smoothly shaded grays at upper center. ASTC's color weight interpolation allows it to handle the smooth gray regions very well, though the 1.07bpp image shows block artifacts as well as a chroma distortion at lower center-right. The very difficult image in the middle row shows characteristic failure modes for the three formats; PVRTC shows blotching on the figure's hat, and severe blurring and quantization at lower center. ASTC at 1.07bpp avoids the quanitzation, but blurs details and shows prominent chrominance block artifacts. ASTC at 2bpp preserves luminance detail fairly well, also at the cost of chrominance blocking. The image in the third row has little luminance variation and is generally smooth. PVRTC performs well except for some stipple-pattern noise in the upper left corner, while ASTC suffers from block artifacts at both bit rates.

Figure 5 shows the same images at a higher bit rate. On the top row, S3TC and (to a lesser extent) ETC2 suffer from color quantization artifacts. ASTC's greyscale and luminance color modes again allow it to capture the smooth gradients very well. In the middle row, PVRTC shows minor blotching on the hat and in the smooth grey regions in the upper half of the image, and some quantization in the noisy regions at lower center. S3TC and ETC2 show minor quantization artifacts; ASTC is able to preserve a little more luminance detail, by giving up some color space resolution in exchange for a high-resolution color weight array. On the third row, PVRTC at 4bpp does very well except for minor stipple noise. S3TC and ETC2 suffer from severe block artifacts. ASTC 6x6 also exhibits block artifacts but at somewhat lower contrast.

5.2. High Dynamic Range Textures

Table 3 presents mPSNR results for some of the images used by Munkberg et al [MCHAM06], measured across the exposure ranges reported in their paper. For comparison, we also present results for BC6H. On average, BC6H is slightly ahead on this dataset, but the differences are small in relative terms.

Figure 6 compares ASTC and BC6H 8bpp encodings on a portion of the 'AtriumNight' and 'Spirals' images. Images are rendered at an exposure of +3. On the AtriumNight image, ASTC preserves slightly more detail. On the Spirals image, both compressed formats exhibit block artifacts, but ASTC also shows luminance distortions. In HDR mode, our encoder attempts to minimize error in the log of image luminance. This causes trouble around pixels with very small values, where the slope of the log function is very large. The result is that the encoder works very hard to minimize errors at very small pixel values, at the expense of fidelity in other parts of the image. We believe that this is a codec issue, and not a fundamental problem with the format.

6. Conclusion

We have described a new texture compression method that spans an extremely wide range of bit rates and use cases. The format introduces several technical innovations, most notably a general method for representing value sequences using a fractional number of bits per value, and a system for constructing per-texel color weights from sparse samples. The resulting image quality is competitive with the most advanced formats in use today, and better than that of industry standards such as DXT and PVRTC.

Acknowledgements

The authors would like to thank Konstantine Iourcha, Cass Everitt, Nick Penwarden, Walt Sullivan, and many others for valuable discussions and feedback.

References

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. 1

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the masses: A hardware rasterization architecture for mobile phones. *ACM Transactions on Graphics (Proc. SIGGRAPH 2003) 22*, 3 (july 2003), 801–808. 1, 2, 3

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 373–378. 1

(a) *Context* **(b)** *Original* **(c)** *PVRTC 2bpp* **(d)** *ASTC 12x10* **(e)** *ASTC 8x8*

Figure 4: *Low bit rate LDR-RGB compression examples. Source images are at left. Detail images, from left to right: Original; PVRTC-2bpp; ASTC 12x10 (1.07 bpp); ASTC 8x8 (2bpp)*

[CDF*86] CAMPBELL G., DEFANTI T. A., FREDERIKSEN J., JOYCE S. A., LESKE L. A.: Two bit/pixel full color encoding. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), SIGGRAPH '86, ACM, pp. 215–223. 2

[DM79] DELP E. J., MITCHELL O. R.: Image compression using block truncation coding. *IEEE Transactions on Communications 27*, 9 (September 1979), 1335–1341. 2

[Fen03] FENNEY S.: Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), HWWS '03, Eurographics Association, pp. 84–91. 2

[FNK94] FRÄNTI P., NEVALAINEN O., KAUKORANTA T.: Compression of digital images by block truncation coding: A survey. *The Computer Journal 37*, 4 (1994), 308–332. 2

[Gus05] GUSTAVSON S.: Simplex noise demystified. Technical report at `http://www.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`, Mar 2005. 5

[INH99] IOURCHA K., NAYAK K., HONG Z.: System and method for fixed-rate block-based image compression with inferred pixel values. US Patent 5,956,431, 1999. 2

[Khr05] KHRONOS: OES_compressed_ETC1_RGB8_texture. available at http://www.khronos.org/registry/gles/, 2005. 2

[Kod] KODAK: Kodak lossless true color image suite. available at `http://r0k.us/graphics/kodak/`. 7

[KSKS96] KNITTEL G., SCHILLING A., KUGLER A., STRASSER W.: Hardware for superior texture performance. *Computers & Graphics 20*, 4 (1996), 475–481. 1

[MCHAM06] MUNKBERG J., CLARBERG P., HASSELGREN J., AKENINE-MÖLLER T.: High dynamic range texture compression for graphics hardware. *ACM Trans. Graph. 25*, 3 (July 2006), 698–706. 7, 8

[NVI] NVIDIA: Nvidia texture tools. available at `code.google.com/p/nvidia-texture-tools/list/`. 7

[Ope10] OPENGL ARB: ARB_texture_compression_bptc. available at http://www.opengl.org/registry/specs/ARB/, 2010. 2

[SAM04] STRÖM J., AKENINE-MÖLLER T.: Packman: texture compression for mobile phones. In *ACM SIGGRAPH 2004 Sketches* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 66–. 2

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), HWWS '05, ACM, pp. 63–70. 2, 5, 7

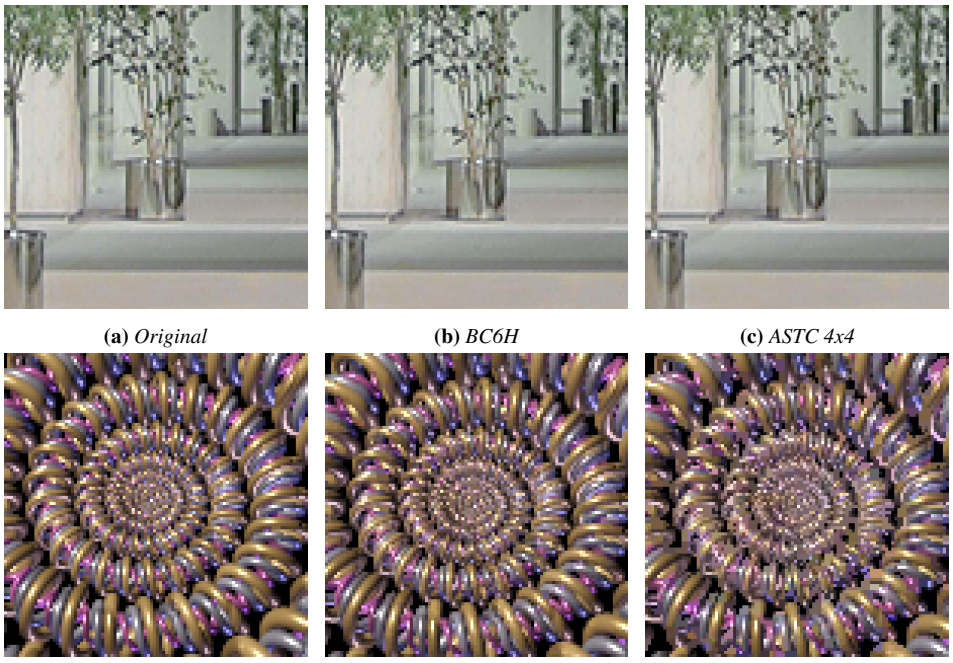
[SP07] STRÖM J., PETTERSSON M.: ETC2: Texture compression using invalid combinations. In *Graphics Hardware* (2007), pp. 49–54. 2, 7

[TK96] TORBORG J., KAJIYA J. T.: Talisman: commodity real-time 3d graphics for the pc. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 353–363. 1

[WS09] WENNERSTEN P., STRÖM J.: Table-based alpha compression. In *Computer Graphics Forum* (2009), vol. 28, Eurographics 2009, pp. 687–695. 3

(a) *Original* **(b)** *PVRTC-4bpp* **(c)** *S3TC* **(d)** *ETC2* **(e)** *ASTC 6x6*

Figure 5: *Medium bit rate LDR-RGB compression examples. From left to right: Original; PVRTC-4bpp; S3TC; ETC2; ASTC 6x6 (3.56bpp)*



(a) *Original* **(b)** *BC6H* **(c)** *ASTC 4x4*

(d) *Original* **(e)** *BC6H* **(f)** *ASTC 4x4*

Figure 6: *HDR image comparisons. From left to right: Original; BC6H (8bpp); ASTC 4x4 (8bpp)*