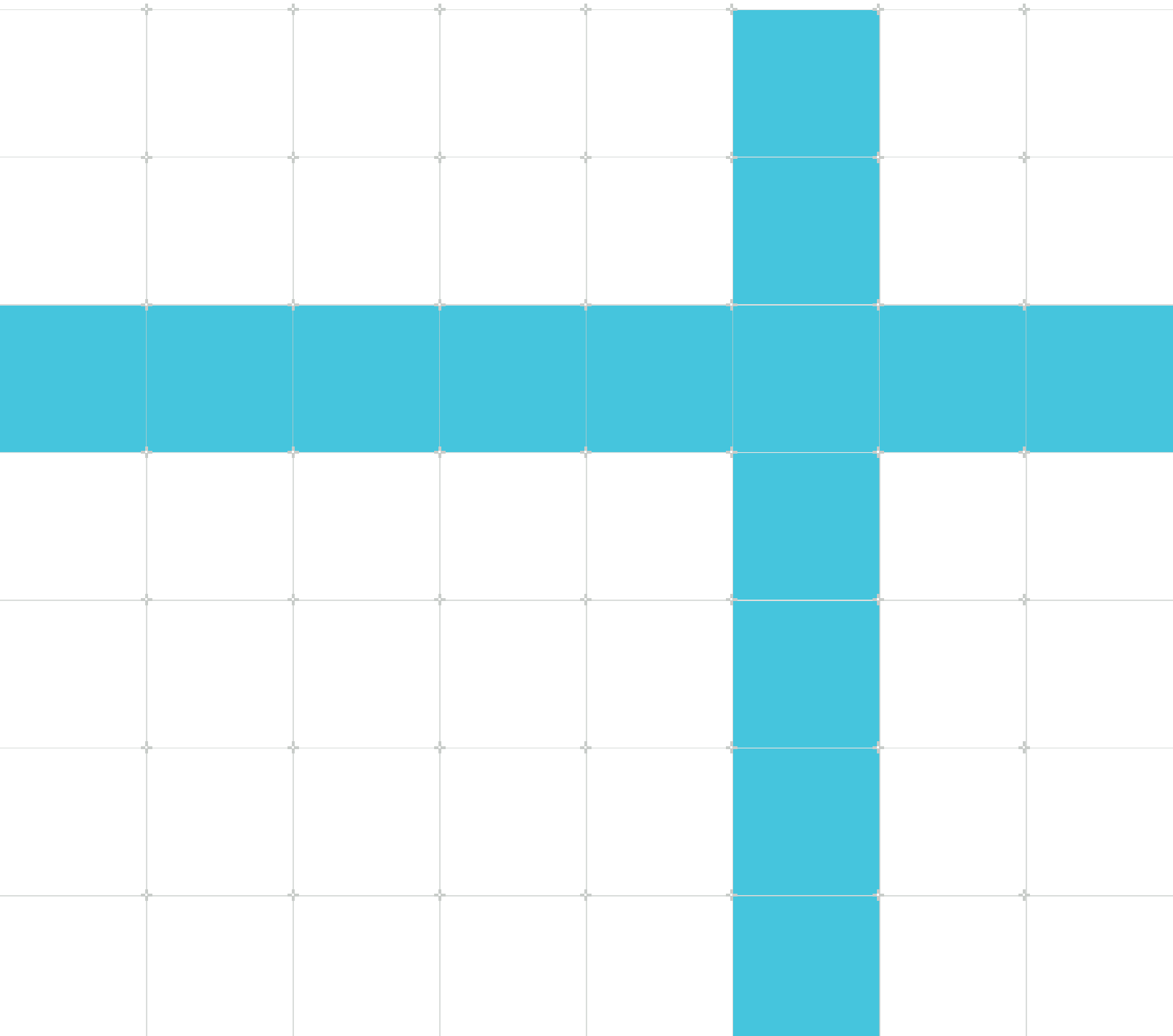
Adaptive Scalable Texture Compression

Version 4.3

Developer Guide

Issue 00
102162_0430_00_en



Adaptive Scalable Texture Compression
Developer Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	1 July 2020	Non-Confidential	First release
0200-00	20 August 2020	Non-Confidential	Second release. Minor updates
0430-00	10 March 2023	Non-Confidential	Covers changes up to 4.3.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

Adaptive Scalable Texture Compression (ASTC) is an advanced lossy texture compression technology developed by Arm® and AMD.

This guide provides information about how you can use ASTC effectively to optimize the performance of your apps. It covers the following subjects:

- What is ASTC and why is it needed?
- Technical details of the ASTC compression algorithm.
- How to use Arm ASTC Encoder (astcenc) to compress game assets.
- How to use ASTC with graphics APIs like OpenGL ES and Vulkan.
- How to use ASTC with the Unity and Unreal Engine gaming engines.

Why use texture compression?

For most content, texture access is one of the main consumers of memory bandwidth in a system. Textures are usually represented as 2D image data. As content texture resolution and texture count increases, so does memory bandwidth. This memory bandwidth overhead can slow down GPU performance if we cannot load data fast enough to keep the shader cores busy. In addition, DRAM access is energy intensive, so high bandwidth also increases power consumption and thermal load.

To reduce the impact on performance, we can use specialized compression schemes to reduce the size of texture resources. These real-time compression schemes are significantly different to the more general types of compression, like JPEG or PNG, that you are probably familiar with.

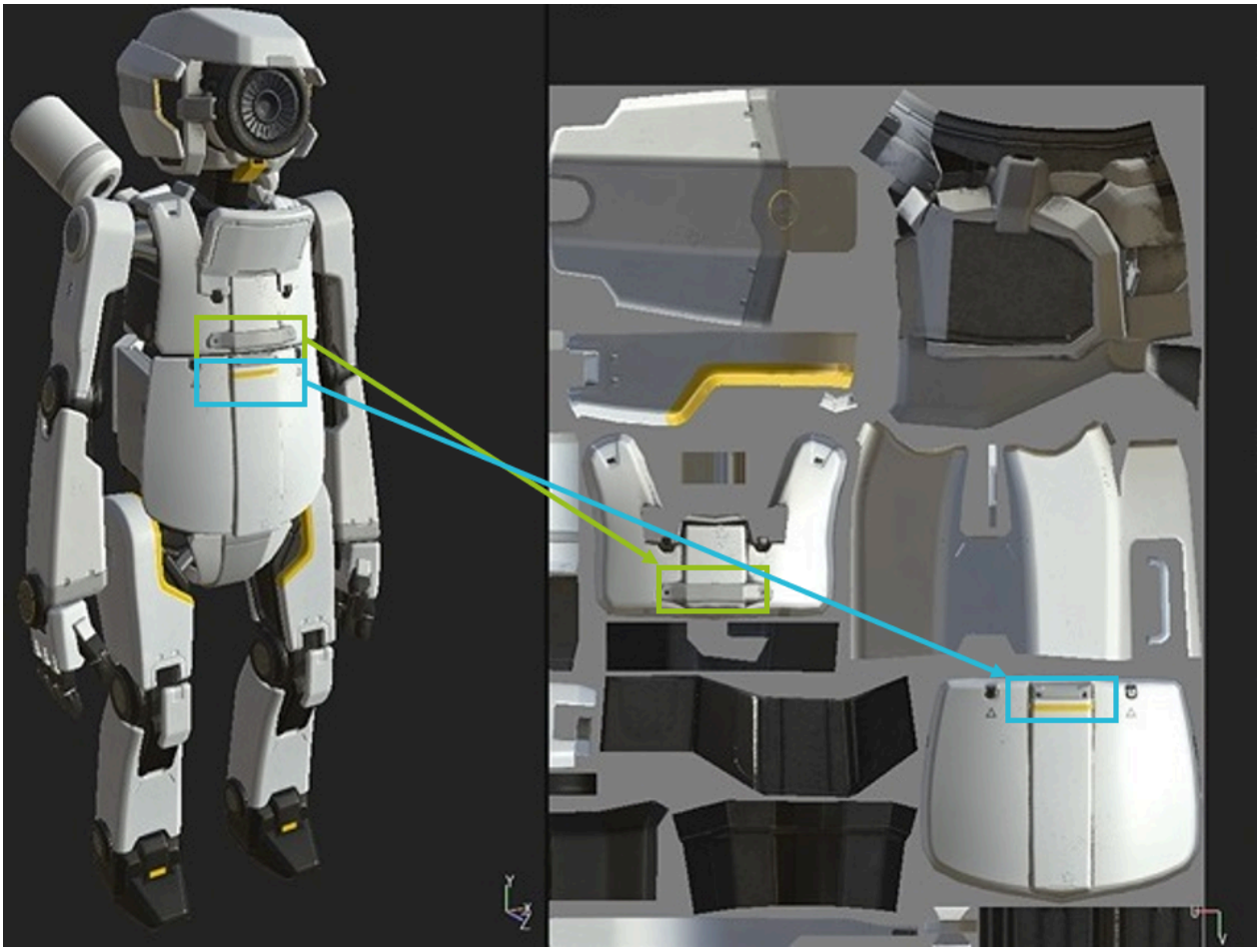
Traditional compression schemes like JPG and PNG are designed to compress or decompress an entire image. They can achieve very good compression rates and image quality. However, they are not designed to let you access smaller portions of the full image easily without decompressing the entire image.

When mapping 2D textures onto a model, individual texels might be extracted from the full texture image in an unpredictable order:

- Not all texture elements might be needed. For example, depending on the orientation of the model, and any other objects that might be obscuring parts of it.
- Texels that are rendered next to each other in the final image may originate from different parts of the texture.

The following image shows the arrangement of different texture elements within a texture image, and a model with that texture applied. Notice that adjacent texels in the rendered image are not necessarily adjacent in the texture image.

Figure 1-1: Robot texture with callouts



It is computationally expensive for a GPU to decompress the entire image when it only needs a small portion of the whole. As a result, real-time compression schemes are designed to provide efficient decoding for random sampling of elements within a larger image, when they are used in shaders.

There are various techniques to achieve this result, but most algorithms do the following:

- Compress a fixed size NxM texel input block
- Write this compressed block out into a fixed number of bits.

This allows simple address calculation in the GPU. Because all input and output sizes are fixed, it is a relatively simple pointer offset calculation to calculate a sample address. We therefore only need to access the data from one NxM block to decompress any single texel.

ASTC format overview

Khronos has adopted ASTC as an official extension to the OpenGL and OpenGL ES APIs, and as a standard optional feature for the Vulkan API. ASTC has the following advantages over older texture compression formats:

Format flexibility

ASTC can compress between one and four channels of data, including one non-correlated channel like RGB+A (correlated RGB, non-correlated alpha).

Bit rate flexibility

ASTC provides a wide choice of bit rates when compressing images, between 0.89 bits and 8 bits per texel (bpt). The bit rate choice is independent of the color format choice.

Advanced format support

ASTC can compress images in either Low Dynamic Range (LDR), LDR sRGB, or High Dynamic Range (HDR) color spaces. ASTC can also compress 3D volumetric textures.

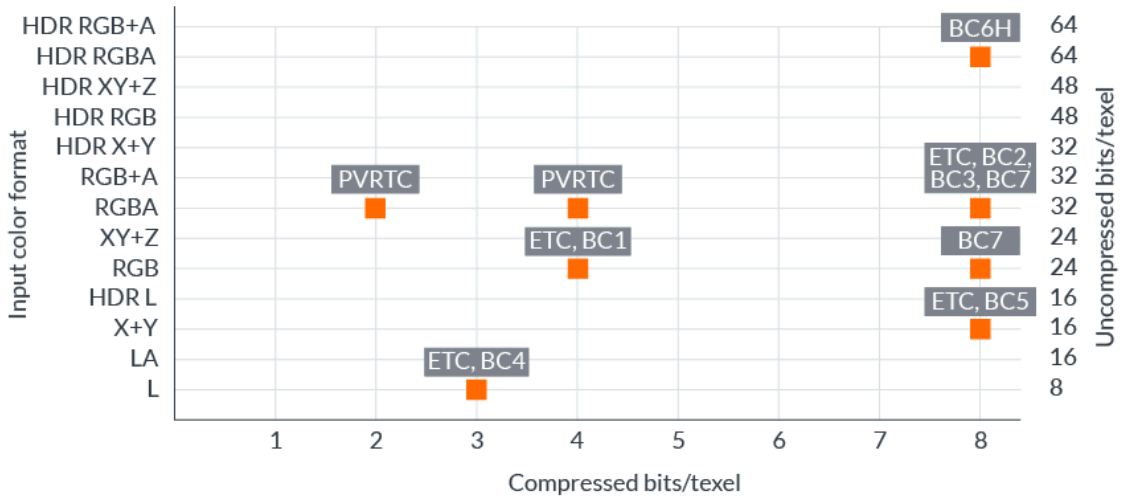
Improved image quality

Despite the high degree of format flexibility, ASTC outperforms nearly all legacy texture compression formats on image quality at equivalent bit rates. Examples of legacy texture compression formats that ASTC outperforms include ETC2 and PVRCT.

Bitrates below 1bpp are achieved by using a system of variable block sizes. Most block-based texture compression methods have a single fixed block size. By contrast, ASTC can store an image with a regular grid of blocks from 4x4 to 12x12, including non-square block sizes. ASTC can also store 3D textures, with block sizes ranging from 3x3x3 to 6x6x6.

Before the creation of ASTC, the available texture compression formats supported relatively few combinations of color format and bit rate, as shown in the following diagram:

Figure 1-2: Legacy coverage

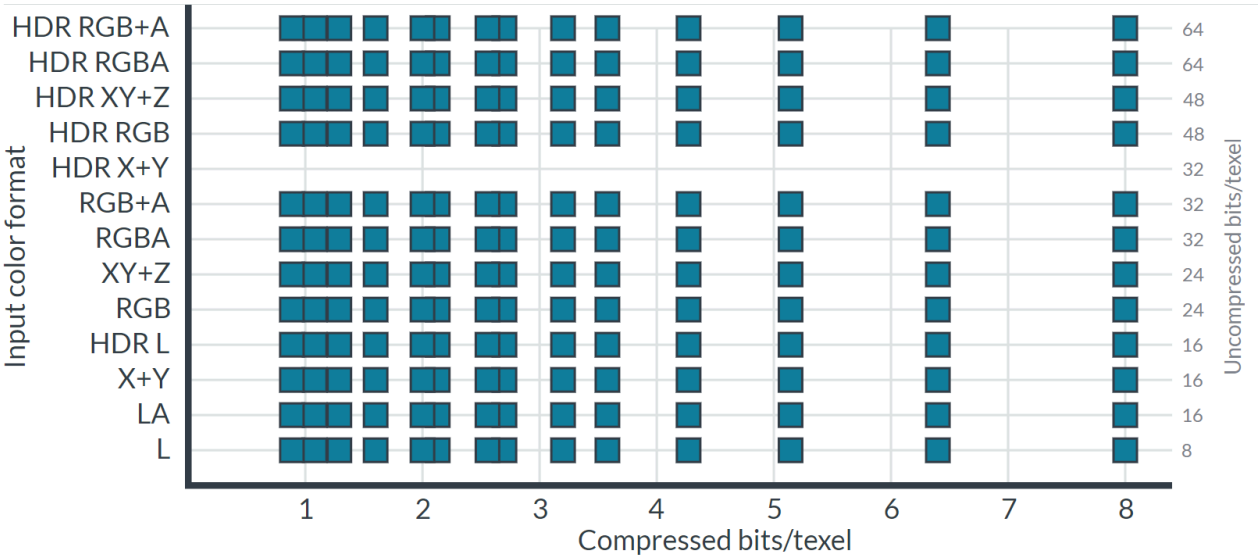


The situation was even worse than this diagram shows. Many formats were either proprietary or not available on some operating systems, so any single platform had very limited compression choices.

This situation made it difficult to develop content that is portable across multiple platforms. Assets might need to be compressed differently for each platform. Each asset pack might need to use different levels of compression, and might have no compression on some platforms. This would leave either some image quality, or some memory bandwidth efficiency, untapped.

A better way was needed. The Khronos group requested proposals for a new compression algorithm to be adopted, like the earlier ETC algorithm that was adopted for OpenGL ES. ASTC was the result, developed by Arm and AMD, and has been adopted as an official algorithm for OpenGL, OpenGL ES, and Vulkan. ASTC achieves almost complete coverage of the desirable format matrix, with a wide choice of bitrates available for content creators. The following diagram shows the available formats and bitrates:

Figure 1-3: ASTC coverage



2. The ASTC format

This section of the guide explains how the ASTC algorithm works. We start with a high-level overview of the block compression algorithm and color encoding process, then we examine the technical details.

Block compression

Compression formats for real-time graphics need the ability to quickly and efficiently make random samples into a texture. This places two technical requirements on any compression format. It must be possible to do the following:

- Compute the address of data in memory given only a sample coordinate.
- Decompress random samples without decompressing too much surrounding data.

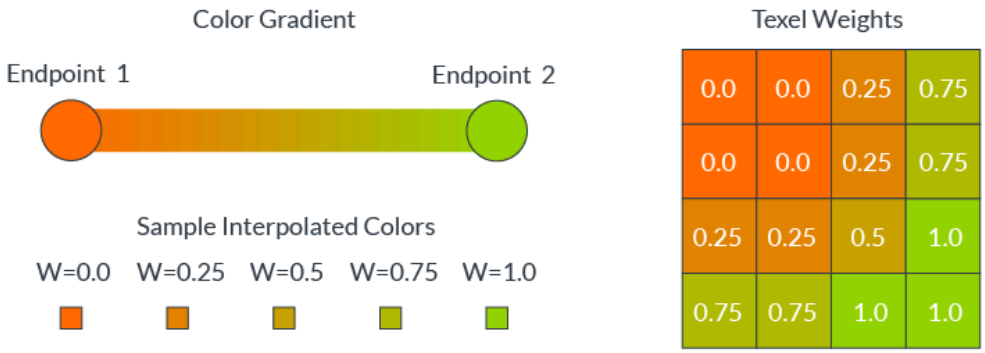
The standard solution that all contemporary real-time formats use, including ASTC, is to divide the image into fixed-size blocks of texels. Each block is then compressed into a fixed number of output bits. This feature makes it possible to access texels quickly, in any order, and with a well-bounded decompression cost.

The 2D block footprints in ASTC range from 4x4 texels up to 12x12 texels, which all compress into 128-bit output blocks. By dividing 128 bits by the number of texels in the footprint, we derive the format bit rates. These bit rates range from 8 bpt (128/(4x4)) down to 0.89 bpt (128/(12x12)).

Color encoding

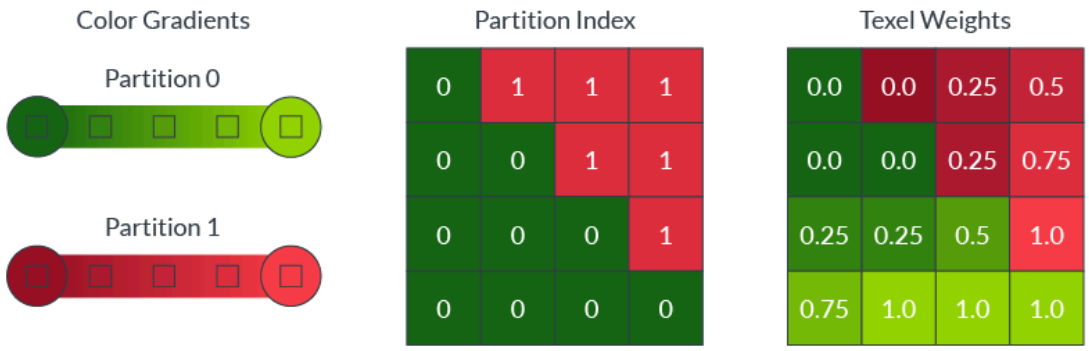
ASTC uses gradients to assign the color values of each texel. Each compressed block stores the end-point colors for a gradient, and an interpolation weight for each texel. During decompression, the color value for each texel is generated by interpolating between the two end-point colors, based on the per-texel weight. The following diagram shows this interpolation for a variety of texel weights:

Figure 2-1: Gradient 1p



Blocks often contain a complex distribution of colors, for example a red ball sitting on green grass. In these scenarios, a single-color gradient cannot accurately represent all the different texel color values. ASTC allows a block to define up to four distinct color gradients, called partitions, and can assign each texel to a single partition. For our example, we require two partitions, one for the red ball texels and one for the green grass texels. The following diagram shows how the partition index specifies which color gradient to use for each texel:

Figure 2-2: Gradient 2p



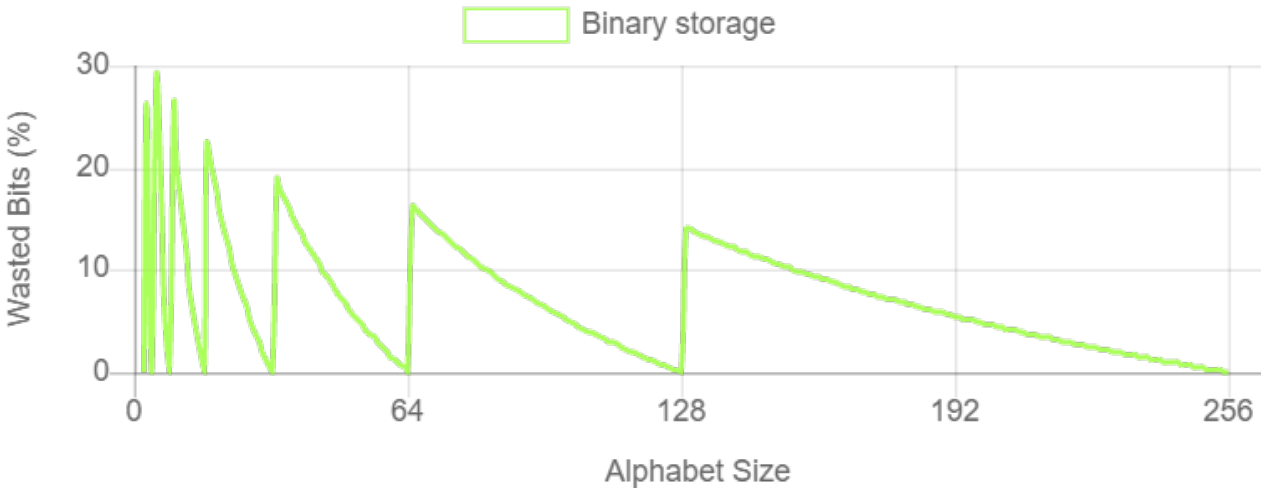
Storing alphabets

Even though color and weight values per texel are notionally floating-point values, we have too few bits available to directly store the actual values. To reduce the storage size, these values must be quantized during compression. For example, if we have a floating-point weight for each texel in the range 0.0 to 1.0, we could choose to quantize to five values: 0.0, 0.25, 0.5, 0.75, and 1.0. We can then represent these five quantized values in storage using the integer values 0-4.

In the general case, if we choose to quantize N levels, we need to be able to efficiently store characters of an alphabet containing N symbols. An N symbol alphabet contains log2(N) bits of information per character. If we have an alphabet of five possible symbols, then each character contains ~2.32 bits of information, but simple binary storage would require us to round up to three bits. This wastes 22.3% of our storage capacity.

The following chart shows the percentage of the bit-space that would be wasted using simple binary encoding to store an arbitrary N symbol alphabet:

Figure 2-3: Binary encoding



This chart shows that for most alphabet sizes, using an integer number of bits per character wastes a lot of storage capacity. Efficiency is critically important to a compression format, so this is an area that ASTC needed to address.

One solution is to round the quantization level up to the next power of two, so that rather than being wasted the extra bits are used to store more accurate data values. However, this solution forces the encoder to spend bits which could be used elsewhere for a bigger benefit, so it reduces image quality and is a suboptimal solution.

Quints and trits

Instead of rounding up a five-symbol alphabet, called a quint, to three bits, a more efficient solution is to pack three quint characters together. Three characters in a five-symbol alphabet have 5^3 (125) combinations, and contain 6.97 bits of information. We can store these three quint characters in seven bits and have a storage waste of only 0.5%.

We can similarly construct a three-symbol alphabet, called a trit, and pack trit characters in groups of five. Each character group has 3^5 (243) combinations, and contains 7.92 bits of information. We can store these five trit characters in eight bits and have a storage waste of only 1%.

Bounded Integer Sequence Encoding

The Bounded Integer Sequence Encoding (BISE) allows storage of character sequences using arbitrary alphabets of up to 256 symbols. Each alphabet size is encoded in the most space-efficient choice of bits, trits, and quints.

- Alphabets with up to $(2^N - 1)$ symbols can be encoded using N bits per character.
- Alphabets with up $3x(2^N - 1)$ symbols can be encoded using n bits (m) and a trit (t) per character, and reconstructed using the equation $((t \times 2^N) + m)$.
- Alphabets with up to $5x(2^N - 1)$ symbols can be encoded using n bits (m) and a quint (q) per character, and reconstructed using the equation $((q \times 2^N) + m)$.

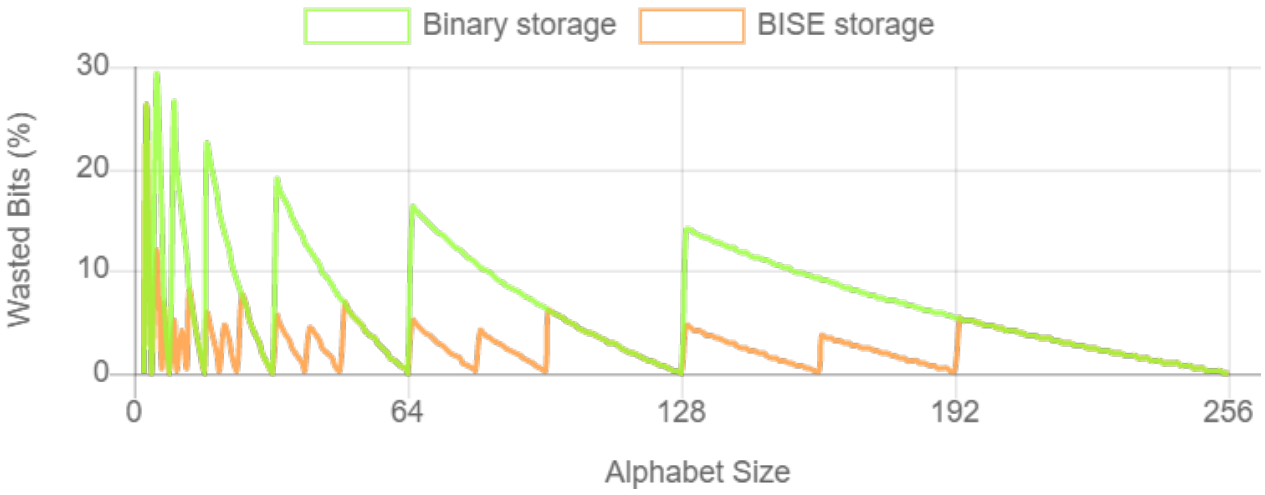
When the number of characters in a sequence is not a multiple of three or five we must avoid wasting storage at the end of the sequence, so we add another constraint on the encoding. If the last few values in the sequence to encode are zero, the last few bits in the encoded bit string must also be zero. Ideally, the number of non-zero bits is easily calculated and does not depend on the magnitudes of the previous encoded values. This is challenging to arrange during compression, but it is possible. This means that we do not need to store any padding after the end of the sequence, because we can safely assume that they are zero bits.

With this constraint in place, and by some smart packing of the bits, trits, and quints, BISE encodes a string of S characters in an N symbol alphabet using a fixed number of bits:

- S values up to (2^N - 1) use (NxS) bits.
- S values up to (3 * 2^N - 1) use (NxS + ceil(8S / 5)) bits.
- S values up to (5 * 2^N - 1) use (NxS + ceil(7S / 3)) bits.

The compressor chooses the option which produces the smallest storage for the alphabet size that is being stored. Some use binary, some use bits and a trit, and some use bits and a quint. If we compare the storage efficiency of BISE against simple binary for the range of possible alphabet sizes that we might want to encode, we see that BISE is much more efficient. The following chart shows the efficiency gain of BISE storage over binary storage:

Figure 2-4: BISE



Block sizes

ASTC always compresses blocks of texels into 128-bit outputs. However, ASTC allows the developer to select from a range of block sizes to enable a fine-grained tradeoff between image quality and size. The following table shows the different block sizes result and the corresponding bits per texel:

Block footprint	Bits per texel
4x4	8.00
5x4	6.40

Block footprint	Bits per texel
5x5	5.12
6x5	4.27
6x6	3.56
8x5	3.20
8x6	2.67
10x5	2.56
10x6	2.13
10x8	1.60
10x10	1.28
12x10	1.07
12x12	0.89

Color endpoints

The color data for a block is encoded as a gradient between two color endpoints. Each texel selects a position along that gradient, which is then interpolated during decompression. ASTC supports 16 color endpoint encoding schemes, known as endpoint modes.

The options for endpoint modes let you vary the following:

- The number of color channels. For example, luminance, luminance+alpha, RGB, or RGBA.
- The encoding method. For example, direct, base+offset, base+scale, or quantization level.
- The data range. For example, low dynamic range or high dynamic range.

The endpoint modes, and the endpoint color BISE quantization level, can be chosen on a per-block basis.

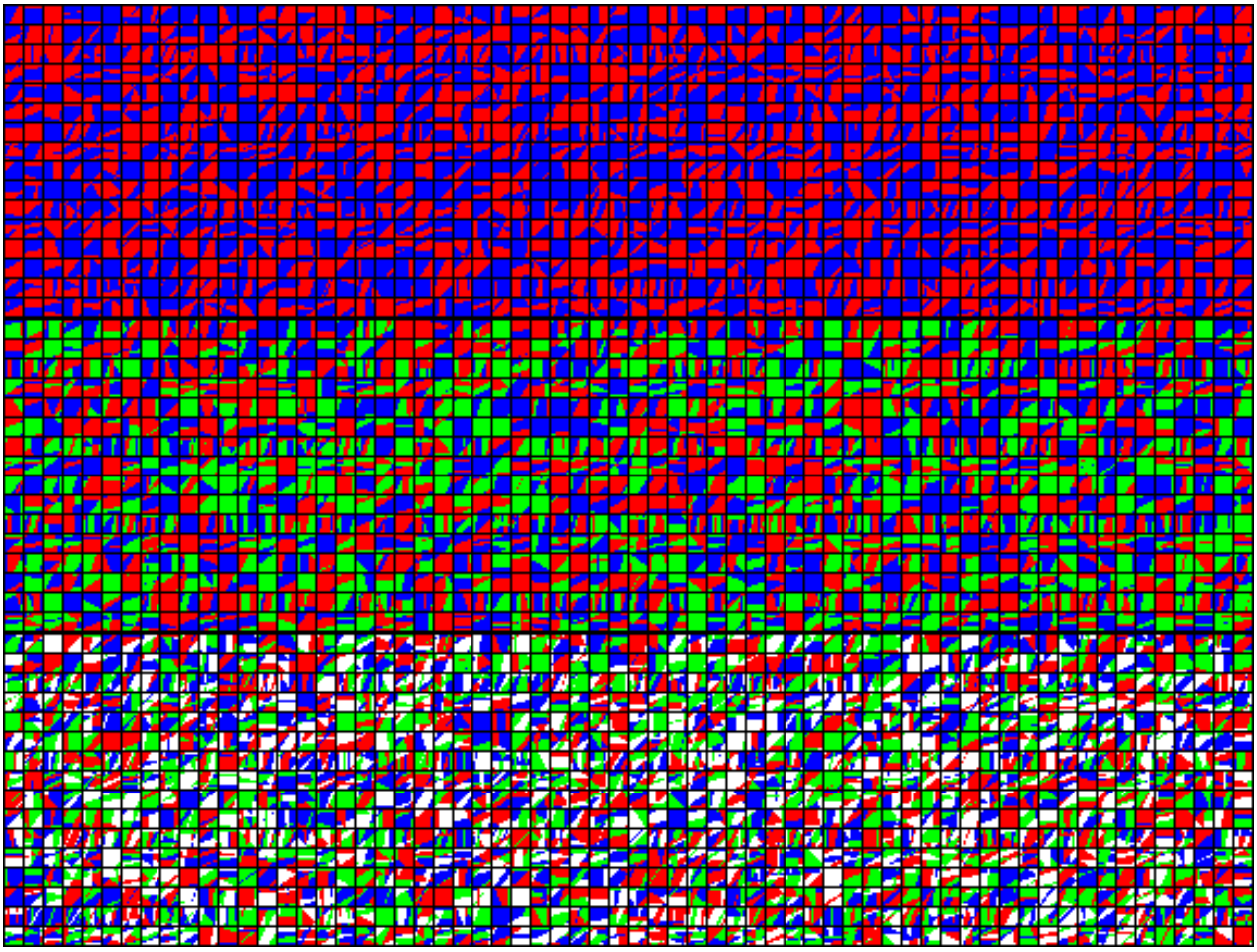
Color partitions

Colors within a block are often complex. As described earlier, a single-color gradient often cannot accurately capture all the colors within a block.

ASTC allows a single block to reference up to four color gradients, called partitions. Each texel is then assigned to a single partition for the purposes of decompression.

Directly storing the partition assignment for each texel would need a lot of decompressor hardware to store it for all block sizes. Instead, ASTC algorithmically generates a range of patterns, using the partition index as a seed value. The compression process selects the best pattern match for each block. The block then only needs to store the index of the best matching pattern. The following image shows the generated patterns for two (top section of image), three (middle section of image), and four (bottom section of image) partitions for the 8x8 block size:

Figure 2-5: Hash



The number of partitions and the partition index can be chosen on a per-block basis, and a different color endpoint mode can be chosen per partition.

Choosing the best encoding for each block

During compression, the algorithm must select the correct distribution pattern and boundary color pairs, then generate the quantized values for each pixel. There is a certain degree of trial and error involved in the selection of patterns and boundary colors, so when compressing there is a trade-off between compression time and final image quality. The higher the quality, the more alternatives the algorithm tries before deciding which is best. However long the compression takes, the decompression time is fixed. This is because the image data can always be re-extrapolated from the pattern and boundary colors in a single pass.

The compression algorithm can use different metrics to judge the quality of different attempts. These metrics range from pure value ratios of signal to noise, to a perceptual judgment weighted towards human visual acuity. The algorithm can also judge the channels individually rather than as a whole. Treating channels individually preserves detail for textures where the individual channels may be used as a data source for a shader program, or to reduce angular noise, which is important for tangent space normal maps.

Texel weights

Each texel requires a weight, which defines the relative contribution of each color endpoint when interpolating the color gradient.

For smaller block sizes, we can choose to store the weight directly, with one weight per texel. However, for the larger block sizes there are not enough bits of storage to do this. To work around this issue, ASTC allows the weight grid to be stored at a lower resolution than the texel grid. The per-texel weights are interpolated from the stored weight grid during decompression using a bilinear interpolation.

Both the number of texel weights, and the weight value BISE quantization level, can be chosen on a per-block basis.

Dual-plane weights

Using a single weight for all color channels works well when there is good correlation across the channels, but this is not always the case. Common examples where we might get low correlation include:

- Textures storing RGBA data. Alpha masks are not usually closely correlated with the color value.
- Textures storing normal data. The X and Y normal values often change independently.

ASTC provides a dual-plane mode, which uses two separate weight grids for each texel. A single channel can be assigned to a second plane of weights, while the other three use the first plane of weights.

The use of dual-plane mode can be chosen on a per-block basis, but its use prevents the use of four-color partitions. This is because there are not enough bits to concurrently store both an extra plane of weights and an extra set of color endpoints.

3. ASTC benefits

ASTC offers several advantages over existing texture compression schemes. We will look at these advantages in this section of the guide.

Adaptive compression

The first word in the acronym ASTC is adaptive, and after reading this guide it should be clear why. Each block always compresses into 128 bits of storage, but the developer can choose from a wide range of texel block sizes. The compressor also gets a lot of freedom to determine how those 128 bits are used.

The compressor can trade off the number of bits assigned to colors and weights on a per-block basis to get the best image quality possible.

Factors that affect the number of bits assigned to colors include:

- the number of partitions
- the endpoint mode
- the stored quantization level

Factors that affect the number of bits assigned to weights include:

- the number of weights per block
- the use of dual-plane
- the stored quantization level

Range of supported formats

The encoding scheme that ASTC uses effectively compresses arbitrary sequences of floating-point numbers, with a flexible number of channels, across any of the supported block sizes. There is no real notion of color format in the compression scheme, beyond the color endpoint mode selection.

This wide range of supported formats and bit rates means that it is possible to compress almost any asset to some degree. You can make appropriate bit rate choices based on quality needs rather than format constraints.

Image quality

The high level of flexibility ASTC provides does not mean that image quality is compromised. On the contrary, an ASTC compressor is not forced to spend bits on things that do not improve image quality. This is made possible by the high packing efficiency allowed by BISE encoding, and the ability to dynamically choose where to spend encoding space on a per-block basis.

This dynamic compression efficiency results in significant improvements in image quality compared to older texture formats, despite ASTC handling a much wider range of options. Comparisons with older formats give the following results:

- ASTC at 2 bpt outperforms PVRTC at 2 bpt by ~2.0dB.

- ASTC at 3.56 bpt outperforms PVRTC and BC1 at 4 bpt by ~1.5dB, and ETC2 by ~0.7dB, despite a 10% bit rate disadvantage.
- ASTC at 8 bpt for LDR formats is comparable in quality to BC7 at 8 bpt.
- ASTC at 8 bpt for HDR formats is comparable in quality to BC6H at 8 bpt.

Differences as small as 0.25dB are visible to the human eye. Remember that dB uses a logarithmic scale, so these results are significant image quality improvements.

3D compression

The techniques in ASTC which underpin the format generalize to compressing volumetric texture data without needing very much extra decompression hardware.

ASTC optionally supports compression of 3D textures, which is a unique feature not found in any earlier format, at the bit rates shown in the following table:

Block footprint	Bits per texel
3x3x3	4.74
4x3x3	3.56
4x4x3	2.67
4x4x4	2.00
5x4x4	1.60
5x5x4	1.28
5x5x5	1.02
6x5x5	0.85
6x6x5	0.71
6x6x6	0.59

4. About Arm ASTC Encoder

The Arm ASTC Encoder (`astcenc`) is an open-source command-line tool for compressing and decompressing ASTC textures.

Get Arm ASTC Encoder from the Arm GitHub repository.

Arm ASTC Encoder supports compression of the following formats into ASTC format output images:

- Low Dynamic Range image formats: BMP, JPEG, PNG, and TGA
- High Dynamic Range image formats: EXR and HDR
- DDS and KTX container formats, although only a subset of format features are supported.

Compressed outputs can be written into a KTX container or a simple `.astc` wrapper.

Use the `-help` option to see usage instructions and details of all available command-line options:

```
astcenc -help
```

Compression

To compress an image, use the `astcenc` command with any required options. Use the `-cl`, `-cs`, `-ch`, and `-cH` modes to select a color profile:

- `-cl` use the linear LDR color profile
- `-cs` use the sRGB LDR color profile
- `-ch` use the HDR color profile, tuned for HDR RGB and LDR A
- `-cH` use the HDR color profile, tuned for HDR RGBA

Use the following options to select the quality level. Note that higher quality compression increases compression time:

- `-fast`
- `-medium`
- `-thorough`
- `-verythorough`
- `-exhaustive`

For most uses we would recommend starting with `-medium` and only moving to higher options for images with insufficient quality.

For example, the following command compresses `example.png` using the LDR color profile and a 6x6 block footprint (3.55 bits/pixel):

```
astcenc -cl example.png example.astc 6x6 -medium
```

The `-medium` quality preset gives a reasonable image quality for a relatively fast compression speed. The output is stored to a linear color space compressed image `example.astc`.

There are many power-user options to control the compressor heuristics. Run `astcenc --help` for more details.

Decompression

Decompress an image using the `-dl`, `-ds`, `-dh`, or `-dH` modes:

- `-dl` uses the linear LDR color profile
- `-ds` uses the sRGB LDR color profile
- `-dh` and `-dH` use the HDR color profile

For example, the following command decompresses `example.astc` using the full HDR feature profile, storing the decompressed output to `example.tga`:

```
astcenc -dh example.astc example.tga
```

Measuring image quality

Review the compression quality using the `-tl`, `-ts`, `-th`, and `-tH` modes:

- `-tl` uses the linear LDR color profile
- `-ts` uses the sRGB LDR color profile
- `-th` uses the HDR color profile, tuned for HDR RGB and LDR A
- `-tH` uses the HDR color profile, tuned for HDR RGBA

For example, the following command uses the LDR color profile and a 5x5 block size to compress `example.png`, using the `-thorough` quality preset, and then immediately decompresses the image and saves the result to `example.tga`:

```
astcenc -tl example.png example.tga 5x5 -thorough
```

This process can be used to enable a visual inspection of the compressed image quality. In addition, this mode also prints out some image quality metrics to the console.

Compressing normal maps

The `astcenc` compressor has a special compression mode for normal maps, as their compression goals are quite different to color data. The normal map mode is enabled by specifying the `-normal` command line option. This has two effects:

- The normal data is assumed to be unit length and packed into four channels as “x x x y”. This allows the compressor to store only two values per endpoint, which frees up a significant amount of bitrate which can be used elsewhere.
- The compressor optimizes for the angular error of the normal vector instead of absolute color channel error, which improves quality of the normal data even if data looks a little worse when viewed as a color image.

As these normal maps only store two components, the Z component of the normal must be reconstructed in shader code based on the knowledge that the vector is unit length. The GLSL code for reconstruction of the Z value is:

```
vec3 normal;
normal.xy = texture(...).ga;
normal.xy = normal.xy * 2.0 - 1.0;
normal.z = sqrt(1 - dot(normal.xy, normal.xy));
```

Compressing 3D textures

To compress a volumetric 3D texture, specify the `-array <size>` option, where `<size>` is the Z dimension of the texture.

When the `-array` argument is specified, the input filename is interpreted as a prefix. The actual input files are named with the specified prefix plus `_0`, `_1`, and so on, up to `<size>-1`.

For example, the following command loads files named `slice_0.png`, `slice_1.png`, `slice_2.png`, and `slice_3.png`:

```
astcenc -c slice.png slice.astc 4x4x4 -array 4 -medium
```

Choosing a suitable bitrate

Choosing a higher or a lower bitrate lets you trade image quality against data size to obtain the optimum balance for your assets.

The following image shows how different bit rates affect image quality:

Figure 4-1: ASTC quality

A good practice is to split all your texture assets into quality categories based on their distance to the viewer or general visibility and importance. For example, you can split your assets into three categories: high, medium, and low. Instead of adjusting the bitrate for each individual texture, experiment with a few textures from each category and determine the best bitrate for each category. You can then use these bitrates to batch-compress the rest of the textures in each category.

For the majority of color textures using a block size between 6x6 (3.56bpp) and 8x8 (2bpp) gives an acceptable quality with efficient memory size and bandwidth.

For 2D user interface elements, where image quality can be more important, a smaller block size such as 4x4 or 5x5 might be more appropriate.

Normal maps compressed with the `-normal` mode option need a higher bitrate than color data. We would recommend using the `-thorough` option to achieve a higher quality compression, and a 6x6 block size.

Compressing sRGB textures

Human perception of light luminosity does not vary linearly in proportion to the actual light luminosity. Our eyes have evolved to be more sensitive to changes between two dark values than between two light values which, in the real world, provides us with improved night vision.

The sRGB color space uses a non-linear gamma curve to generate a value distribution which mimics the perceptual response of the human eye, with more values in the dark range than the bright range. This gives improved perceptual quality for color data when compared to linear data stored at the same bitrate. It is recommended to author color data content in the sRGB color space, and compress it using the ASTC sRGB sub-format. Texture lookups using sRGB textures are converted into linear values during texture sampling, at no runtime cost.

ASTC supports non-linear sRGB color space conversion at both compression and decompression time.

To keep images in sRGB color space until the point that they are used:

1. Compress images with the `-cs` option.
2. When loading images, use the sRGB texture formats instead of the regular texture formats.

   The sRGB texture formats contain `SRGB8_ALPHA8` in the name, for example, `COMPRESSED_SRGB8_ALPHA8_ASTC_4x4_KHR`. There is an sRGB equivalent for every RGBA format.


Note

Do not use sRGB for non-color data, such as normal maps. The perceptual distribution implied by the sRGB gamma curve only makes sense for color data.

Compressing HDR textures

The ASTC format can be used to compress high-dynamic range (HDR) data, giving a similar capability to the BC6 format which is commonly available on consoles.

Compressing HDR data is challenging for the compressor, and best results are achieved using high-bitrate encodings such as 4x4 blocks (8 bpp), or 5x4 blocks (6.4 bpp). Lower bitrates can be used, but are prone to block artifacts in regions with fast luminance changes.

Compressing RGBM textures

The RGBM format provides a way to encode a limited form of high-dynamic range (HDR) RGB data into a low-dynamic range (LDR) texture, using shader code to reconstruct the HDR value. In the encoded data the RGB channels in the texture data store the base color channels, and the alpha channel stores a multiplier scaling factor for the RGB channels.

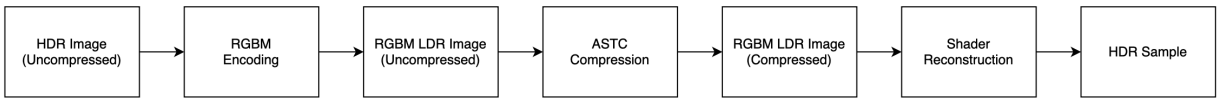

Note

RGBM can only encode data with a moderate dynamic range, for example 0 to 5. This is higher than LDR, but much lower than the full range possible with a true HDR texture format.

The basic workflow for RGBM compression is a two stage process:

1. Convert the HDR source image to an LDR image
2. Use the ASTC compressor to compress the LDR image.

Figure 4-2: RGBM compression workflow



RGBM encoding

The recommended way to encode RGBM data for a pixel, which is handled by the user prior to calling the compressor, is:

```
// Load HDR inputs in range [0, max] and normalize to [0, 1]
float r_in = pixel_in.r / 5.0f;
float g_in = pixel_in.g / 5.0f;
float b_in = pixel_in.b / 5.0f;

// Extract raw RGBM multiplier
float m_enc = max(r_in, g_in, b_in);

// ... but impose a low clamp to help ASTC compression
m_enc = max(m_enc, 16);

// Rescale RGB to fit full range [0, 1] using the multiplier
float r_enc = min(1.0f, r_in / m_enc);
float g_enc = min(1.0f, g_in / m_enc);
float b_enc = min(1.0f, b_in / m_enc);
```

This is a slight modification of the traditional RGBM encoding, adding in an additional limiter to prevent very small M values. This is required because small M values are prone to quantization errors during ASTC compression that cause severe block artefacts. The impact of this limiter is a reduction in the accuracy of very dark colors, although the accuracy is still higher than is achievable with a direct 8-bit LDR encoding.

RGBM compression

RGBM textures should be compressed using the linear LDR mode for ASTC using the `-cl` command line option. Also pass in the max value of M as a parameter via the `-rgbm` command line option, improving image quality by enabling an RGBM-aware error metric.

RGBM reconstruction

Reconstructing the actual HDR color from a sampled RGBM texture requires a small amount of maths in the shader code:

```
vec4 data = texture(sampler, uv);
data.rgb = data.rgb * data.a * 5.0;
data.a = 1.0;
```

5. Using ASTC with graphics APIs

ASTC support is split into a number of feature profiles that are supported by OpenGL ES, OpenGL, and Vulkan, either as part of the core specifications or as extensions.

The ASTC feature profiles are as follows:

- 2D for LDR images
- 2D and sliced 3D for LDR images
- 2D and sliced 3D for LDR and HDR images
- 2D, sliced 3D, and volumetric 3D for LDR and HDR images

The 2D LDR profile is mandatory in OpenGL ES 3.2, and a standardized optional feature for Vulkan. This means that the 2D LDR profile is widely supported on contemporary mobile devices. The 2D HDR profile is not mandatory, but is widely supported.

Khronos OpenGL ES support and extensions

The LDR profile of ASTC is part of the core API for OpenGL ES 3.2. Support for earlier API versions and for the other format features is provided via a number official Khronos extensions:

- `KHR_texture_compression_astc_ldr`: 2D for LDR images
- `KHR_texture_compression_astc_sliced_3d`: 2D and sliced 3D for LDR images
- `KHR_texture_compression_astc_hdr`: 2D and sliced 3D for LDR and HDR images

There is also an extension that provides the full feature set implied by supporting all three KHR extensions, with the addition of the volumetric 3D texture support. This extension provides the 3D block sizes, such as 4x4x4, allowing the compressor to exploit coherency across planes in the image to improve quality or reduce bitrate:

- `OES_texture_compression_astc`: 2D + 3D, LDR + HDR support

ASTC supports sliced 3D textures, where each 2D slice is compressed independently, as well as volumetric 3D, which compresses the whole image using a volumetric block footprint such as the 4x4x4 block size. The `OES_texture_compression_astc` extension is the only extension that adds support for the volumetric 3D block sizes, and is a superset of the KHR extensions.

Decode mode extensions

ASTC decompresses texels into fp16 intermediate values, except for sRGB which always decompresses into 8-bit UNORM intermediates. For many use cases, this gives more dynamic range and precision than required, and can cause a reduction in texturing efficiency due to the larger data size.

The following extensions, supported on recent mobile GPUs, allow applications to reduce the intermediate precision to either UNORM8 or RGB9e5:

- `GL_EXT_texture_compression_astc_decode_mode`: Allow UNORM8 intermediates.
- `GL_EXT_texture_compression_astc_decode_mode_rgb9e5`: Allow RGB9e5 intermediates.

UNORM8 is recommended for LDR textures, and RGB9e5 is recommended for HDR textures.

For more information about using the ASTC decode mode extension to select decoding precision when decoding ASTC image blocks, see the OpenGL ES SDK for Android: ASTC low precision tutorial.

Vulkan support and extensions

The 2D LDR profile of ASTC is a core feature in Vulkan, but supporting the format it is optional. Platform support can be queried at runtime by testing the `textureCompressionASTC_LDR` physical device feature.

The HDR profile is provided by the following extension:

- `VK_EXT_texture_compression_astc_hdr`

Decode mode extension

As with OpenGL ES, by default Vulkan implementations are required to decompress non-sRGB ASTC data to 16-bit floating point values. However, you can change this behavior and choose lower precision using the Vulkan decode mode extension, which covers both LDR and HDR formats:

- `VK_EXT_astc_decode_mode`

6. Unity and ASTC

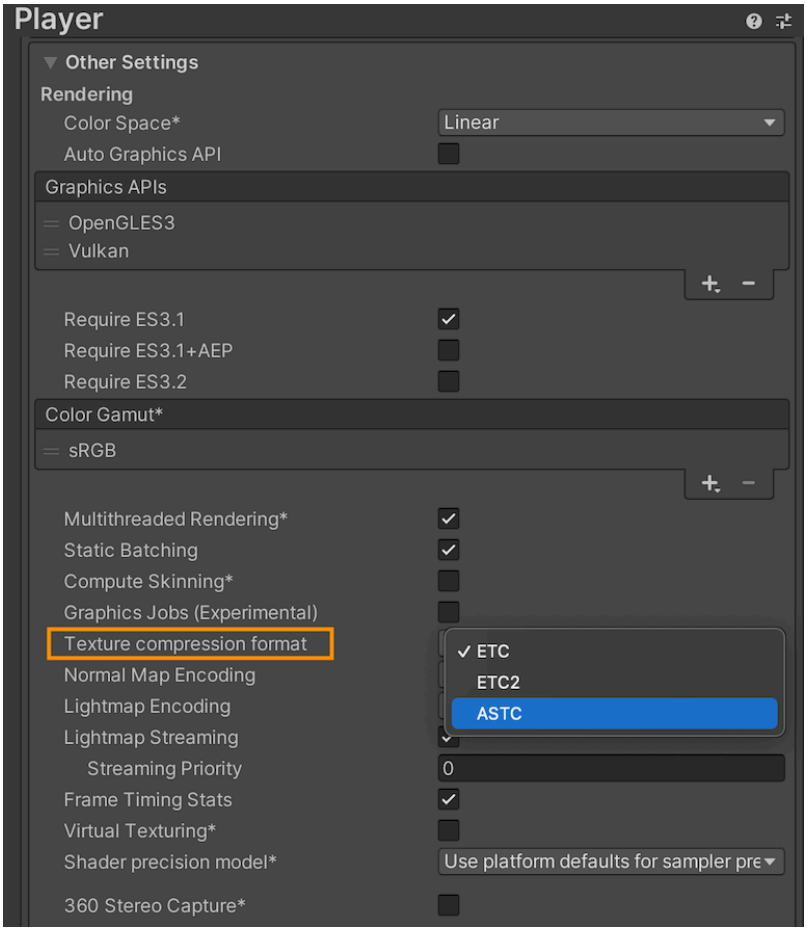
Unity lets you enable ASTC on Android, iOS, tvOS, and WebGL if the hardware supports it. On Android, both HDR and LDR profiles are available.

Each platform has default texture compression formats. For Android, the defaults are ETC2 for RGBA textures and ETC for RGB textures.

Player settings

You can select the default texture commpression format for the project in Unity's Player Settings. Go to Edit > Project Settings > Player Settings > Other Settings.

Figure 6-1: Unity Player settings

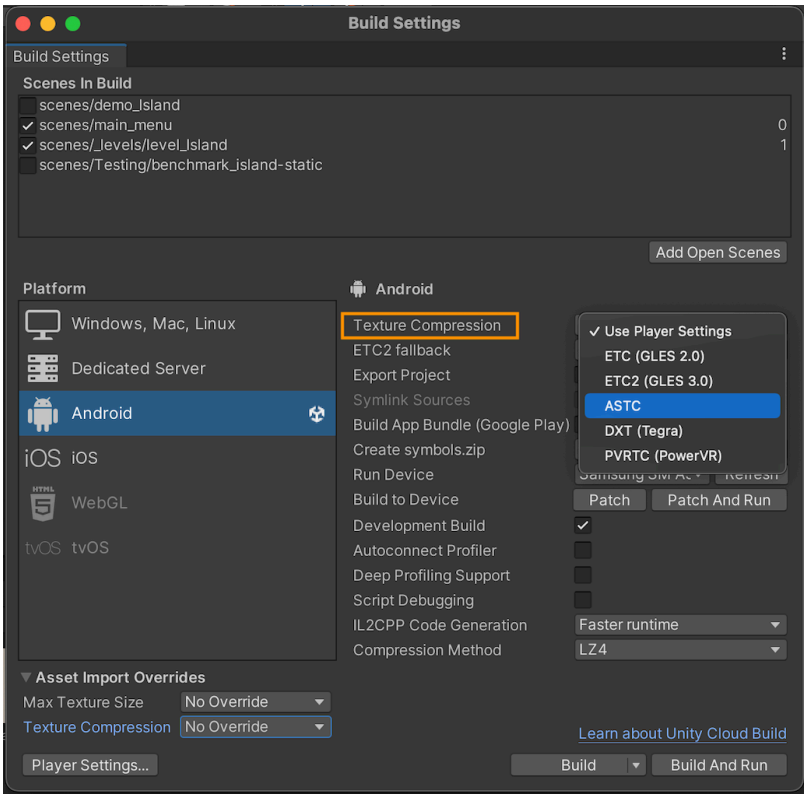


The Texture Compression format setting is global for all textures. This setting can be overridden for specific builds, in Unity's Build settings, or for specific textures, in the Texture settings for each texture.

Build settings

You can set the compression setting for a specific build in the Build settings. Go to File > Build Settings.

Figure 6-2: Unity build settings

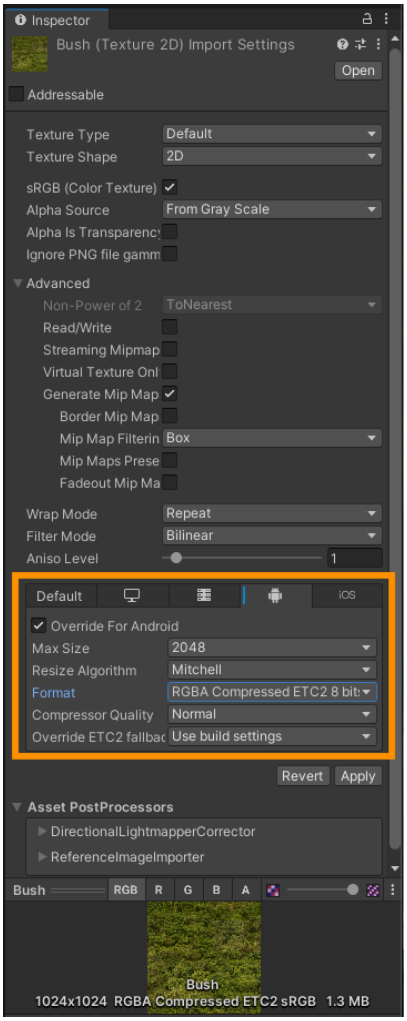


The default is to use the texture compression format specified in the Player settings.

Texture-specific settings

Use the Override for platform section in Texture settings to override the global compression format for an individual texture.

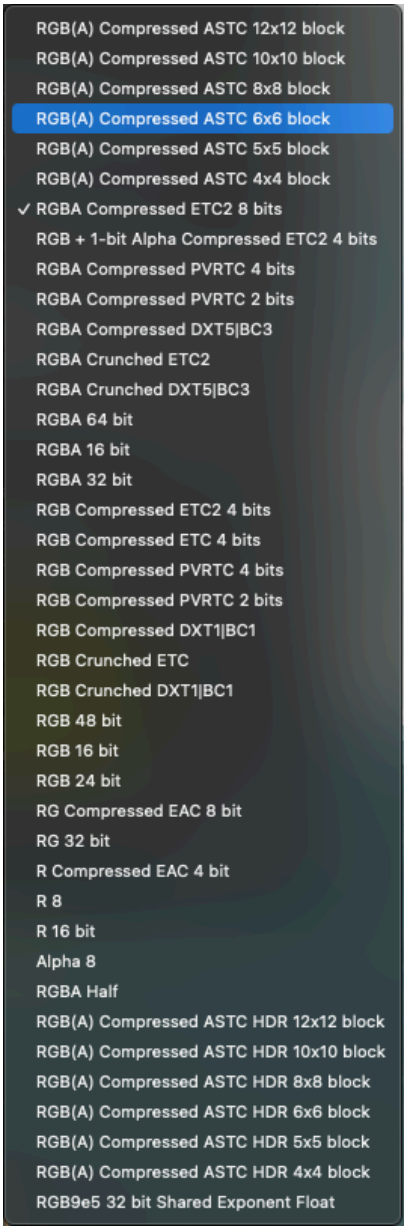
Figure 6-3: Unity texture override



If you select Best for Compression Quality, the compressor thoroughly chooses the optimum ASTC blocks, but at the cost of increased compression time.

You can specify the size of ASTC blocks by choosing the corresponding Format.

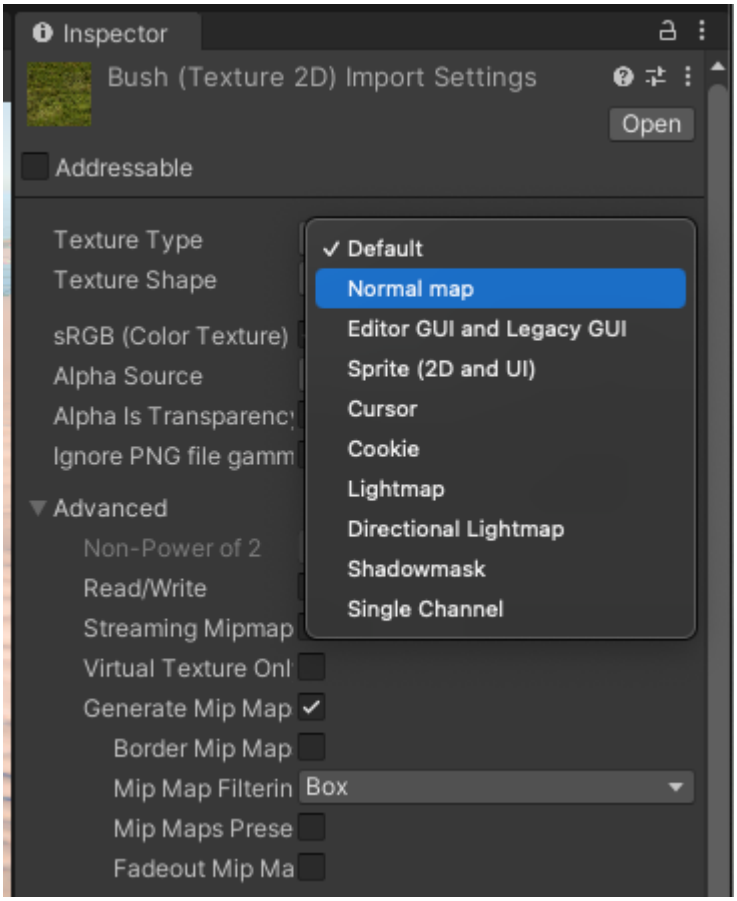
Figure 6-4: Unity formats



If Override for platform is disabled, the settings in this section are still useful. You can consult these settings to see which format and settings are in use for this specific texture, based on the global setting and texture type.

The Import Settings for each texture let you specify texture type and alpha channel information. These settings enable Unity choose appropriate compression settings for the texture.

Figure 6-5: Unity texture importer



Make sure you select Normal map if needed.

After you apply the texture settings, Texture Preview at the bottom of the Inspector lets you see the result and the selected compression format and memory usage:

Figure 6-6: Unity texture preview

Building for different hardware

Not every device supports ASTC. If you have chosen a texture format your platform does not support, texture data is decompressed at runtime. This does not have a huge impact on loading time because Unity uses robust decoders. However, you lose all the other benefits of texture compression like lower memory footprint and cache efficiency.

To avoid this situation, you can build separate APKs for different hardware. For example, ASTC is available on devices with OpenGL ES 3. For OpenGL ES 2, you can use ETC.

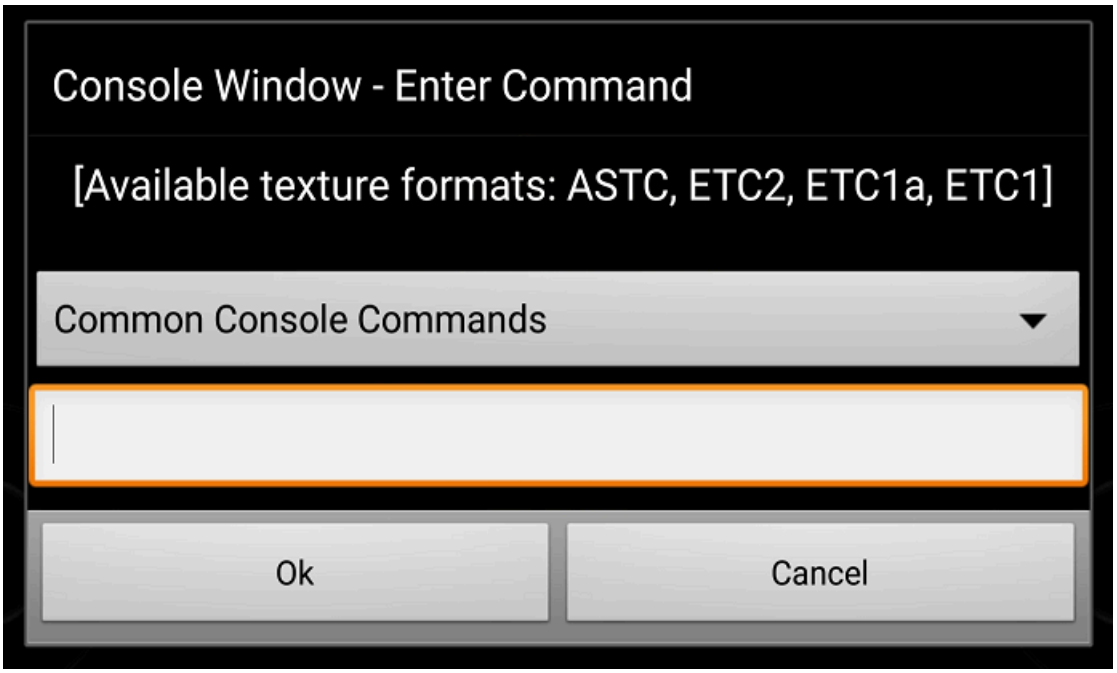
For example, the following steps show one of the possible workflows that you can follow to build different APKs for different platforms:

1. Build for OpenGL ES 3 and Vulkan:
    a. Select ASTC in File > Build Settings for Android.
    b. Go to Edit > Project Settings > Player Settings and disable OpenGL ES 2.
    c. Build the APK.
2. Build for OpenGL ES 2:
    a. Select Use Player Settings in File > Build Settings for Android.
    b. Ensure that all the texture-specific overrides for the platform are either disabled or use ETC format.
    c. Go to Edit > Project Settings > Player Settings and leave only OpenGL ES 2 enabled.
    d. Build the APK.

7. Unreal Engine and ASTC

Unreal Engine supports ASTC for mobile devices and other formats. To see which formats your device supports, tap with four fingers when running the game. The Console Window appears, showing the available texture formats as shown in the following image:

Figure 7-1: Unreal Engine supported formats

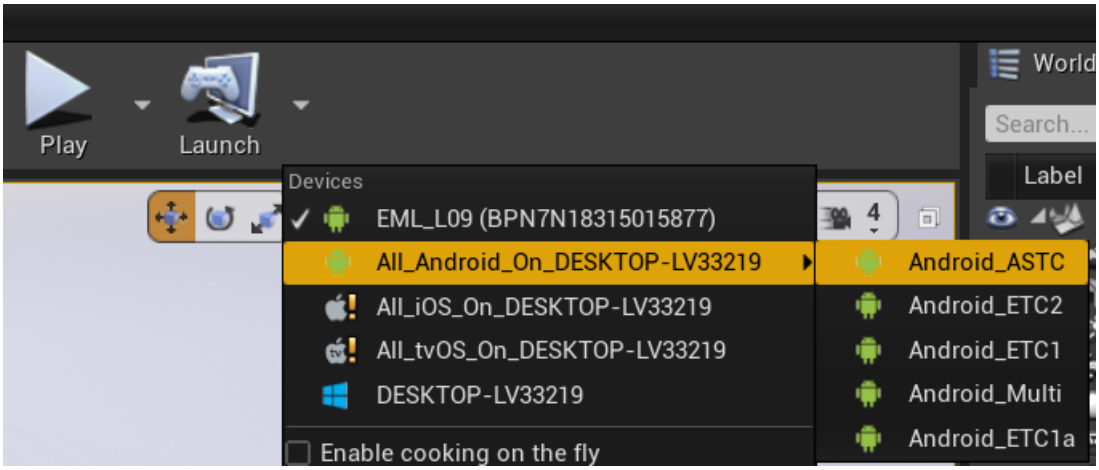


Note

The bShowConsoleOnFourFingerTap variable controls this behavior. The bShowConsoleOnFourFingerTap variable is disabled by default with the Shipping build configuration.

Open the Launch context menu to see the available options for texture compression:

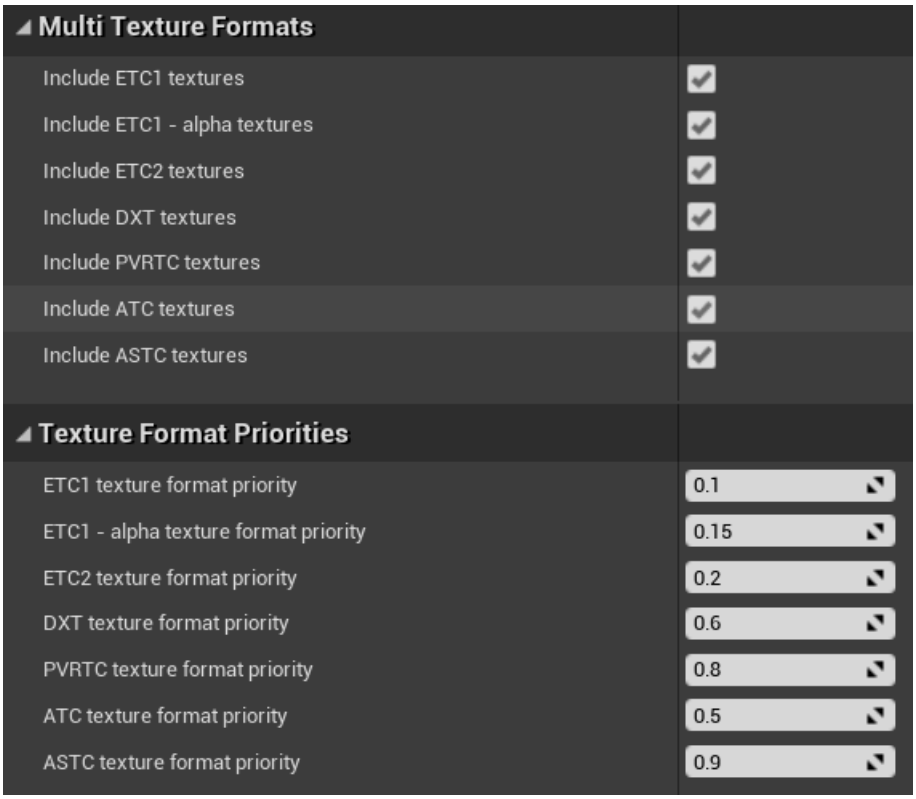
Figure 7-2: Unreal Engine launch options



The Multi option compresses textures with all available formats. This option increases package size and build time but guarantees that the best available format is chosen at runtime. ASTC can be unsupported on some devices, but ETC1 support is guaranteed.

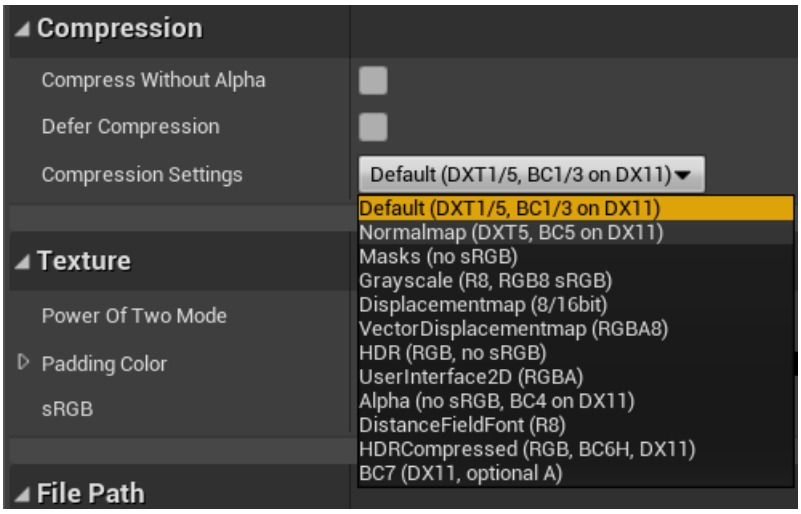
In Project Settings, you can choose which formats are included in the Multi build variant and set their relative priorities, as shown in the following image:

Figure 7-3: Unreal Engine multi texture settings



For normal maps, masks, and GUI elements, ensure that you specify the appropriate compression settings in the Texture Editor, as you can see here:

Figure 7-4: Unreal Engine texture editor



8. Related information

Here are some resources related to material in this guide:

- Download
  - Get ASTC from GitHub
- Blog posts:
  - ASTC Texture Compression: Arm Pushes the Envelope in Graphics Technology
  - How low can you go? Building low-power, low-bandwidth Arm Mali GPUs
- Graphics API ASTC examples
  - ASTC and OpenGL ES SDK example
  - ASTC and Vulkan SDK example
  - OpenGL ES SDK for Android: ASTC low precision tutorial
- Presentations:
  - ASTC: The Future of Texture Compression
- Technical documentation:
  - Khronos OpenGL extensions
- Video:
  - Get the Most out of Adaptive Scalable Texture Compression

9. Next steps

This guide introduced the fundamental principles of what ASTC is, how it evolved, and how to use ASTC in your applications.

The next steps are to start using the ASTC format to compress the texture assets in your own projects.

If you are using a gaming engine to develop your project, you can learn more about using ASTC with these resources:

- Unreal Engine users: Arm Guide for Unreal Engine 4: Optimizing Mobile Gaming Graphics
- Unity users: Arm Guide for Unity Developers: Optimizing Mobile Gaming Graphics

Finally, to see a demonstration of the results you can achieve with ASTC, watch the Arm Mali ASTC Texture Compression Demo from CES 2014.