Mali SDK:
“Texture Compression and Alpha Channels”

Date of Issue:	28 November 2011
Product:	Mali SDK
Product Version:	1.0.0

Abstract

This document describes the related samples “ETCAtlasAlpha”, “ETCCompressedAlpha, and “ETCUncompressedAlpha”, which illustrate three different ways of handling alpha channels when using ETC1 compression.

Release Information

Proprietary Notice

Document Confidentiality Status

ARM Web Address

The ARM website is located at the following address: http://www.arm.com

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions, please submit them to the Mali Developer Center forum www.malideveloper.com.

Feedback on this document

If you have any comments on or about this document, submit them to the Mali Developer Center forum www.malideveloper.com.

General suggestion for additions and improvements are also welcome.

Texture Compression and Alpha Channels

Introduction

This document is a guide to getting the most out of your partially transparent texture maps when developing software for the Mali GPU, by using hardware texture compression.

Why use texture compression?

Just as the compression on a JPEG image allows more images to fit on a disk, texture compression allows more textures to fit inside graphics hardware, which on mobile platforms is particularly important. The Mali GPU has built in hardware texture decompression, allowing the texture to remain compressed in graphics hardware and decompress the required samples on the fly.

On the Mali Developer site there is a texture compression tool for compressing textures into the format recognised by the Mali GPU, the Ericsson Texture Compression format.

The ETC1 standard

Ericsson Texture Compression version 1 or ETC1 is an open standard supported by Khronos and widely used on mobile platforms. It is a lossy algorithm designed for perceptive quality, based on the fact that the human eye is more responsive to changes in luminance than chrominance.

One minor problem with this standard however is that textures compressed in the ETC1 format lose any alpha channel information, and can have no transparent areas. As there are quite a few clever things that can be done using alpha channels in textures, this has led many developers to use other texture compression algorithms, many of which are proprietary formats with limited hardware support.

This document and associated code samples show several methods of getting transparency into textures compressed with the ETC1 standard.

The Basics

The Mali Developer code samples already contain an example of loading an ETC1 texture and associated pre-scaled mipmaps, so for clarity the snippets for these methods, and the full examples provided to illustrate a full implementation, are based on that simple ETC1 loading example.

Extracting the alpha channel

The first step in any of these methods is extracting the alpha channel from your textures. Since the alpha channel is not packed in the compressed texture, it has to be delivered alongside it. The alpha channel can be extracted with most graphics programs, but since performing that task would be quite arduous this functionality is integrated into the ARM Mali Texture Compression Tool (from version 3.0). Whether, and how, the alpha channel is extracted may be selected by choosing an Alpha handling option the Compression Options dialog. This is shown in Figure 1.

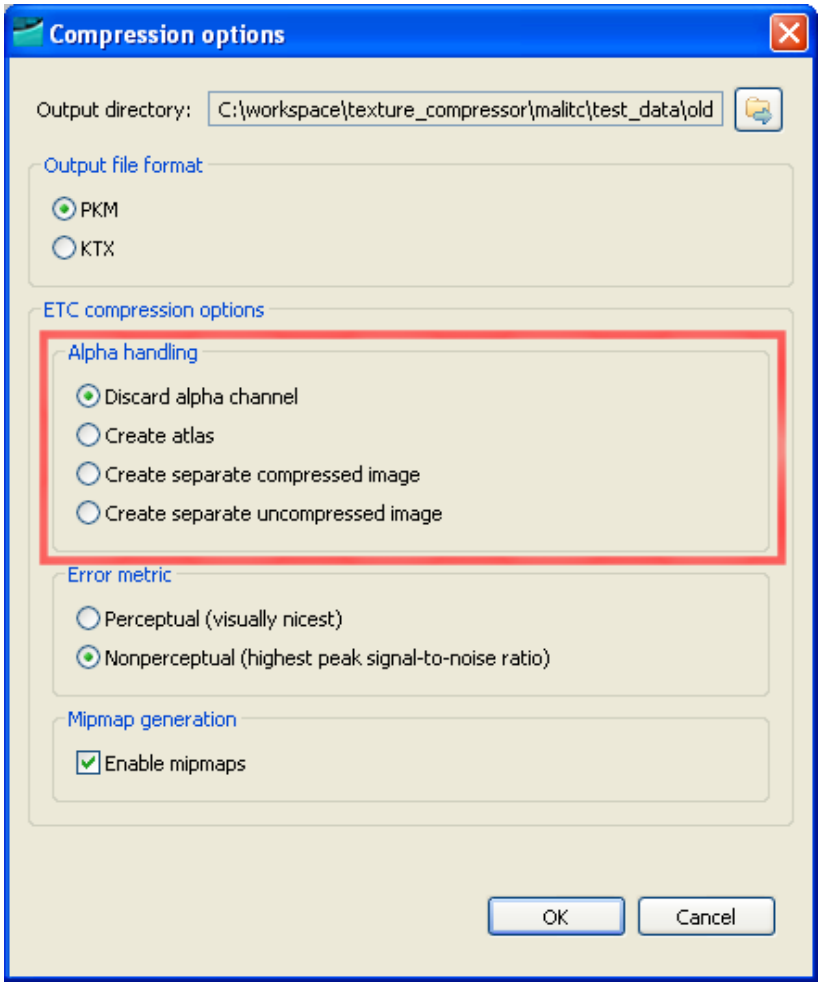


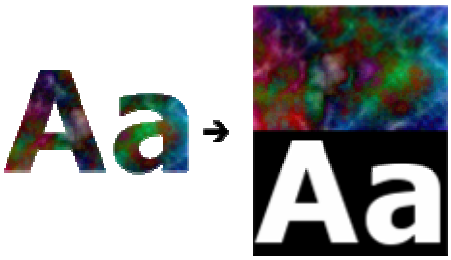
Figure 1 Alpha handling options in the Texture Compression Tool

The Texture Compression Command Line Tool also supports extracting the alpha channel. For full information about using these options see the Texture Compression Tool User Guide.

Method 1: Texture Atlas

Summary

The alpha channel is converted to a visible greyscale image, which is then concatenated onto the original texture, making the texture graphic taller.



Benefits

Still only one file (minimal changes to texture loading procedure)

Only change to code needed is scaling in the shader code

Drawbacks

Texture samples will only wrap properly in one direction

Scaling slows down shader execution

Method

To create compressed images suitable for use with this method select “Create atlas” in the Texture Compression Tool.

The sample “ETCAtlasAlpha” reads an image using this method.

This is the easiest method to implement as when the texture atlas image has been compressed, the only change required in your code is a remapping of texture coordinates in the shader, such that:

```
gl_FragColor = texture2D(u_s2dTexture, v_v2TexCoord);
```

Becomes:

```
vec4 colour = texture2D(u_s2dTexture, v_v2TexCoord * vec2(1.0, 0.5));
colour.a = texture2D(u_s2dTexture,
            v_v2TexCoord * vec2(1.0, 0.5) + v_v2TexCoord * vec2(0.0, 0.5)).r;
gl_FragColor = colour;
```

This scales texture coordinates to use the top half and then shifts that down to the bottom half of the image for a second sample where the alpha channel is. This example uses the red channel of the image mask to set the alpha channel.

More practically, you could instead add a second varying value to your vertex shader.

Making your vertex shader look like this:

```
attribute vec4 a_v4Position;
attribute vec2 a_v2TexCoord;
varying vec2 v_v2TexCoord;
varying vec2 v_v2AlphaCoord;

void main()
{
      v_v2TexCoord = a_v2TexCoord * vec2(1.0, 0.5);
      v_v2AlphaCoord = v_v2TexCoord + vec2(0.0, 0.5);
      gl_Position = a_v4Position;
}
```

And your fragment shader can then use the two varying coordinates:

```
uniform sampler2D u_s2dTexture;
varying vec2 v_v2TexCoord;
varying vec2 v_v2AlphaCoord;

void main()
{
      vec4 colour = texture2D(u_s2dTexture, v_v2TexCoord);
      colour.a = texture2D(u_s2dTexture, v_v2AlphaCoord).r;
      gl_FragColor = colour;
}
```

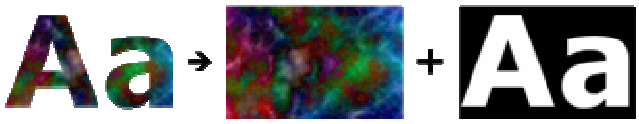
This uses a little more bandwidth for the extra varying vec2, but makes better use of pipelining, particularly as most developers tend to do more work in their fragment shaders than their vertex shaders. With these minor changes most applications should run just fine with the texture atlas files.

However, there are cases when you might want to maintain the ability to wrap a texture over a large area. For that there are the other two methods, discussed below.

Method 2: Separately Packed Alpha

Summary

The alpha channel is delivered as a second packed texture, both textures are then combined in the shader code.



Benefits

More flexible, allows alpha/colour channels to be mixed and matched

Allows for texture wrapping in both directions

Drawbacks

Requires a second texture sampler in the shader.

Method

To create compressed images suitable for use with this method select “Create separate compressed image” in the Texture Compression Tool.

The sample “ETCCompressedAlpha” reads an image using this method.

When loading the second texture be sure to call `glActiveTexture(GL_TEXTURE1)` before `glBindTexture` and `glCompressedTexImage2D` in order to ensure that the alpha channel is allocated in a different hardware texture slot.

```
glActiveTexture(GL_TEXTURE0);
loadCompressedMipmaps(TEXTURE_FILE, TEXTURE_FILE_SUFFIX, &iTexName);
glActiveTexture(GL_TEXTURE1);
loadCompressedMipmaps(ALPHA_FILE, TEXTURE_FILE_SUFFIX, &iAlphaName);
```

When setting your shader uniform variables, you’ll need to allocate a second texture sampler and bind it to the second texture unit:

```
iLocSampler = glGetUniformLocation(iProgName, "u_s2dTexture");
glUniform1i(iLocSampler, 0);
iLocSamplerAlpha = glGetUniformLocation(iProgName, "u_s2dAlpha");
glUniform1i(iLocSamplerAlpha, 1);
```

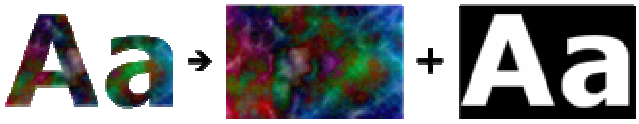
Then inside your fragment shader, once again merge the two samples, this time from different textures:

```
vec4 colour = texture2D(u_s2dTexture, v_v2TexCoord);
colour.a = texture2D(u_s2dAlpha, v_v2TexCoord).r;
gl_FragColor = colour;
```

Method 3: Separate Raw Alpha

Summary

The alpha channel is provided as a raw 8 bit single-channel image, combined with the texture data in the shader.



Benefits

More flexible, allows alpha/colour information to be mixed and matched.

Allows uncompressed alpha, in case lossy ETC1 compression caused artefacts.

Drawbacks

Requires a second texture sampler in the shader

Uncompressed alpha takes up more space than compressed (although still far less than an uncompressed RGBA texture)

Method

To create compressed images suitable for use with this method select “Create separate compressed image” in the Texture Compression Tool.

Depending what other options are selected this will produce either single a PGM file, or a PGM file for each mipmap level, or a single KTX file containing all mipmap levels as uncompressed data. PGM format is described in http://netpbm.sourceforge.net/doc/pgm.html. KTX format is described in http://www.khronos.org/opengles/sdk/tools/KTX/file_format_spec/.

You’ll need to implement a new method to load and bind this texture, but given the uncompressed nature of the texture loading is fairly trivial:

The sample “ETCCompressedAlpha” reads an image using this method.

```
FILE *pFile = NULL;
unsigned char *pTexData = NULL;
unsigned int iWidth = 0;
unsigned int iHeight = 0;
size_t result = 0;

pFile = fopen(pFilename, "rb");

// Read the header. The header is text. Fields are magic number, width, height,
// maximum gray value.
// See http://netpbm.sourceforge.net/doc/pgm.html for format details.

unsigned int iRange = 0;
int iReadCount = fscanf(pFile, "P5 %d %d %d", &iWidth, &iHeight, &iRange);
if(iReadCount != 3) {
    LOGE("Error reading file header of %s", pFilename);
    exit(1);
}
if(iRange != 255) {
    // We can only handle a maximum gray/alpha value of 255, as generated by
    // the Texture Compression Tool
    LOGE("Alpha file %s has wrong maximum gray value, must be 255", pFilename);
    exit(1);
}
```

```
// Read and thow away the single header terminating character
fgetc(pFile);

pTexData = (unsigned char *)calloc(iWidth * iHeight, sizeof(unsigned char));
result = fread(pTexData, sizeof(unsigned char), iWidth * iHeight, pFile);
if (result != iWidth * iHeight)
{
   LOGE("Error reading %s", pFilename);
   exit(1);
}
GL_CHECK(glGenTextures(1, pTexName));
GL_CHECK(glBindTexture(GL_TEXTURE_2D, *pTexName));
GL_CHECK(glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, iWidth, iHeight,
         0,GL_LUMINANCE,GL_UNSIGNED_BYTE, pTexData ));
GL_CHECK(glGenerateMipmap(GL_TEXTURE_2D));
free(pTexData);
```

Allowing the textures to be loaded into separate active textures like before:

```
glActiveTexture(GL_TEXTURE0);
loadCompressedMipmaps(TEXTURE_FILE, TEXTURE_FILE_SUFFIX, &iTexName);

glActiveTexture(GL_TEXTURE1);
loadRawLuminance(ALPHA_FILE, &iAlphaName);
```

And once again loading them separately into the fragment shader:

```
iLocSampler = glGetUniformLocation(iProgName, "u_s2dTexture");
glUniform1i(iLocSampler, 0);
iLocSamplerAlpha = glGetUniformLocation(iProgName, "u_s2dAlpha");
glUniform1i(iLocSamplerAlpha, 1);
```

Then inside your fragment shader merge the two samples, again from the two different textures:

```
vec4 colour = texture2D(u_s2dTexture, v_v2TexCoord);
colour.a = texture2D(u_s2dAlpha, v_v2TexCoord).r;
gl_FragColor = colour;
```