Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-03-17</td>
<td>Rev. 0.10, P. Kirst, O. Wohlmuth</td>
</tr>
<tr>
<td></td>
<td>First draft including review of each others chapters</td>
</tr>
<tr>
<td>2011-03-22</td>
<td>Review by A. von Treuberg</td>
</tr>
</tbody>
</table>

This document contains 15 pages.
Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Semiconductor Europe GmbH restricts its warranties and its liability for all products delivered free of charge (eg. software include or header files, application examples, target boards, evaluation boards, engineering samples of IC’s etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Semiconductor Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling.

Note, all these products are intended and must only be used in an evaluation laboratory environment.

1. Fujitsu Semiconductor Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days form the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Semiconductor Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.

2. Should a Product turn out to be defect, Fujitsu Semiconductor Europe GmbH’s entire liability and the customer’s exclusive remedy shall be, at Fujitsu Semiconductor Europe GmbH’s sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Semiconductor Europe GmbH in original packing and without further defects resulting from the customer’s use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Semiconductor Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Semiconductor Europe GmbH.

3. To the maximum extent permitted by applicable law Fujitsu Semiconductor Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.

4. To the maximum extent permitted by applicable law, Fujitsu Semiconductor Europe GmbH’s and its suppliers’ liability is restricted to intention and gross negligence.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES

To the maximum extent permitted by applicable law, in no event shall Fujitsu Semiconductor Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect.
## Contents

REVISION HISTORY ............................................................................................................ 2

WARRANTY AND DISCLAIMER ......................................................................................... 3

CONTENTS .......................................................................................................................... 4

1 INTRODUCTION.............................................................................................................. 6

2 GENERAL OPTIMIZATION ............................................................................................. 7
  2.1 Make use of the display controllers layer concept.................................................... 7
  2.2 Use 2D graphics operations instead of 3D graphics operations where possible ...... 7
  2.3 Use multiple threads for graphic content with different update requirements ........... 7
  2.4 Use the non-blocking sync mechanisms provided by the graphics driver ............... 8
  2.5 Enable warnings in the release version of the graphics driver to get warnings for inefficient application code ........................................................................................... 8

3 2D - PIXEL ENGINE......................................................................................................... 9
  3.1 Optimize Color Format .......................................................................................... 9
  3.2 RLD sources ......................................................................................................... 9
  3.3 Use combined blit operations ................................................................................ 9
  3.4 Use nearest filter if no bilinear filtering is required ................................................. 9
  3.5 Reduce background redraw area .......................................................................... 9
  3.6 Generic hints for complex scenes ....................................................................... 9

4 3D - OPENGL ES 2.0..................................................................................................... 10
  4.1 General .................................................................................................................. 10
    4.1.1 Minimize OpenGL state changes ................................................................. 10
      4.1.1.1 What causes an OpenGL state change? ....................................................... 10
      4.1.1.2 How to minimize OpenGL state changes? .................................................. 10
    4.1.2 Use vertex buffer objects (VBO's) ............................................................... 11
    4.1.3 Simplify meshes ............................................................................................ 11
    4.1.4 Use LOD (Level of Detail) management ....................................................... 11
    4.1.5 Only draw objects that are visible .................................................................. 11
    4.1.6 Do only clear buffers that need to be cleared .............................................. 11
    4.1.7 Use the Pixel Engine to draw background images instead of textured triangles .......................................................................................................................... 11
    4.1.8 Use triangle strips/fans instead of triangles .................................................. 11
    4.1.9 Choose as low precision as possible (without rendering artefacts) on vertex, normal, color and texture coordinates ................................................................. 11
    4.1.10 Interleave vertex data (attributes) in memory ............................................ 12
4.1.11 Use glDrawArray()'s with different offsets instead of glVertexPointer() and glDrawArray() with offset 0................................. 12
4.1.12 Disable depth test if not needed.................................................... 12
4.1.13 Enable triple buffering................................................................. 12

4.2 Optimizing Shader Programs.......................................................... 12
4.2.1 Use dedicated shader programs instead of universal shaders ........ 12
4.2.2 Avoid if-else constructs in the shader.......................................... 13
4.2.3 Only perform calculations in the shader program that need to be done at run time ................................................................. 13
4.2.4 Carefully design your lighting..................................................... 13
4.2.5 Minimize the number of (temporary) variables............................. 13
4.2.6 Don't use too many assigns......................................................... 13
4.2.7 Don't allocate variables across a large code range, i.e. minimize their lifetime................................................................. 13
4.2.8 Make sure attribute precision in vertex shader and precision of vertex data in memory is identical............................................. 13
4.2.9 Use mediump precision instead of highp (especially fragment shader)..... 13
4.2.10 Perform vector based operations if possible.................................. 14
4.2.11 Use multiplication instead of division........................................... 14

4.3 Optimizing Textures ................................................................. 14
4.3.1 Make your texture images as small as possible.............................. 14
4.3.2 Use a color format that requires less bpp (Bits Per Pixel).............. 14
4.3.3 Use Mipmaps............................................................................ 14
4.3.4 Draw objects sorted by texture.................................................... 14
4.3.5 Check if texturing is really needed............................................. 15
4.3.6 Use per quad texel access for supersampled scenes (compiler option).... 15
1 Introduction

This document provides a collection of useful hints on how to optimize the graphics performance of an application running on MB86R1X "Emerald-X". The first chapter provides general hints that are independent of the used graphics API (2D or 3D). This is followed by chapters that focus on the 2D API (Pixel Engine) respectively 3D API (OpenGL ES 2.0).
2 General Optimization

This chapter discusses optimizations that are independent of the API used for drawing. They are based on the following general rules on how to improve graphics performance:

- Minimize memory bandwidth consumption
- Only redraw your graphics when needed
- Minimize the area that needs to be (re)drawn

2.1 Make use of the display controllers layer concept

The display controllers in MB86R1X ‘Emerald-X’ support multiple layers (see the MB86R1X ‘Emerald-X’ HW manual for a detailed description of the layer concept). In the MB86R1X ‘Emerald-X’ graphics driver, a display controller layer corresponds to a window (see the graphics driver Display API for details). Consider the following aspects when using window(s) (layer and window are interchangeable in the following section):

- Make your window(s) are as small as possible. They don't need to cover the full screen.
- Use separate windows if you have graphics content with different update rates. Example: One layer contains a speedometer that is redrawn with 60 fps and another layer holds the status bar that gets redrawn with 1 fps.
- Carefully use overlapping windows: Overlapping windows increase the required memory bandwidth as the display controller has to fetch the pixel data of all windows and blend them together real-time during display refresh. Overlapping windows make sense if you have
  - high redraw rates. In this case, blending the images together before display would result in a higher memory bandwidth.
  - display regions with different redraw rates. This includes static background images with dynamic content in the front.
  Otherwise it's better to blend the graphics together in one window before it gets displayed.
- Use the minimal color depth required for your application. This minimizes the memory bandwidth required to display the window. Every window can have a different color depth.

2.2 Use 2D graphics operations instead of 3D graphics operations where possible

2D graphics operations are almost always faster than 3D graphics operations. Therefore use 2D instead of 3D graphics operations where possible. Please note that 2D and 3D graphics operations can also work on the same framebuffer.

2.3 Use multiple threads for graphic content with different update requirements

Put all graphics calls for the window(s) with the same redraw requirement, i.e. frame rate, into one thread. Use multiple threads if you have different redraw requirements. By setting the thread priority and the graphics command queue priority (see function mmIGdcConfigSetAttribute() in the graphics driver) you can make sure real-time graphics content is handled with higher priority than non-realtime graphics content.
Consider to limit the frame rate by setting the swap interval (2D: \texttt{mmlGdcDispSyncWindowSwap()}, 3D: \texttt{elgSwapInterval()}). This can be used to get a constant frame rate and/or better distribution of graphics performance among threads.

\textbf{Important Note:} The current version of the graphics driver does not yet support multithreaded OpenGL ES 2.0 applications.

### 2.4 Use the non-blocking sync mechanisms provided by the graphics driver

In a graphics application there should never be the need to wait for the 2D or 3D drawing to be finished by calling \texttt{mmlGdcPeFinish()} or \texttt{glFinish()}. These calls wait until all 2D/3D graphics operations have been finished which result in 2D/3D pipeline bubbles. Try to use the non-blocking sync mechanism instead (see graphic driver Sync API for details).

### 2.5 Enable warnings in the release version of the graphics driver to get warnings for inefficient application code

The release version of the graphics driver prints out warnings for inefficient application code (e.g. precision of vertex shader attributes is different from precision of vertex data in memory). The print out of warnings has to be enabled in the graphics driver (see graphic driver Error Reporting API for details). Note that the production version of the graphics driver doesn’t contain these checks!
3 2D - Pixel Engine

This chapter discusses optimizations for applications using the Pixel Engine API. Please note that some of the following hints influence the processing speed directly (like filter settings) but most hints influence the required memory bandwidth and in this case not only the 2D pixel processing speed but also the memory bandwidth left over for a parallel running 3D processing.

3.1 Optimize Color Format

Use the smallest possible color format for source (and frame buffers). For instance alpha bitmaps can be defined as 8 bit per pixel (bpp) buffers without color information. In many cases even 4, 2 or 1 bpp bitmaps can be used without visible artefacts.

In some cases also reduced customised color formats are sufficiently. For instance an image without alpha channel and a constant red value can be defined as 16 bpp format with 8 bits for green and 8 bits for blue values.

3.2 RLD sources

Try to use run length encoded images if the processing properties allow it (rotation and scaling is not possible for RLD sources).

3.3 Use combined blit operations

For instance a render process with the following steps:
- clear frame buffer
- copy background to frame buffer
- blend new rotated needle over the background

can be realised with a single blit operation:
- blend the rotated needle over the background source and write the result to the frame buffer

3.4 Use nearest filter if no bilinear filtering is required

For instance a rotated alpha source used to define the transparency for a scale does not require bilinear filtering.

3.5 Reduce background redraw area

Use the PixEng driver API to read back the draw area of the last blit operation and refresh only the affected area for the next frame.

3.6 Generic hints for complex scenes

Especially for scene with complex 2D content the hints using layers and threads are important. The tutorial example 2d_threading shows different ways to control the render speed of different render jobs by
- Using more than 2 render buffers to avoid wait operations
- Change the swap interval (function mmlGdcDispSyncWindowSwap()) for layers
- Change the graphics command queue priority (mmlGdcConfigSetAttribute() with parameter MML_GDC_CONFIG_ATTR_GFX_PRIORITY)
- Change the user mode thread processing priority
4 3D - OpenGL ES 2.0

This chapter discusses optimizations for applications using the OpenGL ES 2.0 API.

4.1 General

4.1.1 Minimize OpenGL state changes

With every OpenGL state change,
1. The complete rendering pipeline has to be flushed
2. The new state is set
3. Rendering continues using the new OpenGL state

These pipeline flushes slow down rendering significantly.

4.1.1.1 What causes an OpenGL state change?

Following is a list of OpenGL calls that cause a pipeline flush. Only the most commonly used calls are mentioned:

- `glBlendXXX()`
- `glClear()`
- `glDepthXXX()`
- `glEnable(), glDisable()`
- `glFinish()`
- `glFrontFace()`
- `glScissor()`
- `glTexParameterXX()`
- `glTexImage2D()`
- `glTexParameterXX()`
- `glUseProgram()`

4.1.1.2 How to minimize OpenGL state changes?

- Render objects together that use the same rendering state, e.g.
  - Render objects together that use the same texture (avoids flushing the texture cache)
  - Render objects together that use the same shader program (avoids changing shader programs).
  - Render objects together that use identical fragment operations.
  - Position meshes of complex objects already in model space. This avoids changing the MVP (ModelViewProjection) matrix between meshes. An example would be a car: You can position the parts of a car (doors, tyres, rims etc.) using the MVP matrix or have one model which has the meshes
already positioned correct. Only use the MVP matrix if meshes of an object need to move independently from each other.

- Try to group several texture images into one texture image so texture changes are minimized.

### 4.1.2 Use vertex buffer objects (VBO's)

VBO's reside in VRAM and are copied there only once at load time. In contrast, for objects drawn without VBO's the vertex data has to be copied from CPU memory to VRAM every time the object is drawn.

### 4.1.3 Simplify meshes

Although this is an obvious hint, it is mentioned here as its performance gain is very high. Usually the polygon count of models can be drastically reduced without impacting the visual appearance.

### 4.1.4 Use LOD (Level of Detail) management

Use simplified models (lower polygon count) when objects are far away from the viewer. As objects get closer, switch to more detailed models.

### 4.1.5 Only draw objects that are visible

In some scenes, it is easy to determine at application level if an object is completely hidden or not. Although no pixels are drawn for such objects, they still have to pass the vertex and fragment processing stage (fragment processing only if the triangle is not thrown away after vertex processing). If objects can't be eliminated from rendering, think about the following ways to minimize the number of triangles/fragments that need to be drawn:

- Use scissoring
- Use backface culling

### 4.1.6 Do only clear buffers that need to be cleared

Usually there is no need to clear the color buffer as the complete frame is redrawn.

### 4.1.7 Use the Pixel Engine to draw background images instead of textured triangles

The Pixel Engine is much faster as it does a simple memory copy operation instead of feeding all pixels through the 3D rendering pipeline.

### 4.1.8 Use triangle strips/fans instead of triangles

In triangle strips/fans, vertices are shared between adjacent triangles. These shared vertices are only processed once by the vertex shader.

### 4.1.9 Choose as low precision as possible (without rendering artefacts) on vertex, normal, color and texture coordinates

Lower precision data requires less memory space, i.e. less memory bandwidth to read it by the GPU. Usually lower precision data is processed faster by the shader. See “Optimizing Shader Programs” for details.
4.1.10 Interleave vertex data (attributes) in memory

Vertex attributes (vertex coordinates, normals, texture coordinates etc.) can be put into separate buffers or interleaved into one single buffer. Interleaved data can be read more efficiently from memory as it is one continuous data stream. Following is an example to illustrate the difference between interleave/non-interleaved data:

```c
/* Interleaved vertex data */
typedef struct {
    GLfloat x,y,z; /* vertex coordinates */
    GLfloat nx,ny; /* normal vector */
    GLfloat s,t;   /* texture coordinate */
} v_t;

v_t v_data0[] = {...};

/* Non-interleaved vertex data */
GLfloat v_data1_vertex[]  = {...};  /* vertex coordinates (x,y,z) */
GLfloat v_data1_normal[]  = {...};  /* normal vector (nx, ny) */
GLfloat v_data1_texture[] = {...};  /* texture coordinates (s, t)*/
```

4.1.11 Use glDrawArray()'s with different offsets instead of glVertexPointer() and glDrawArray() with offset 0

The vertex attributes for different objects drawn with glDrawArray() can be put into separate buffer or one single buffer. By using different buffer offsets, these different objects can be drawn without the need to switch buffer. If no OpenGL state change happens between this glDrawArray() calls, rendering is faster using a single buffer.

4.1.12 Disable depth test if not needed

In some situations the depth test can be disabled which results in better graphics performance. Common situations where the depth test can be disabled are:

- A mesh is always on top of the other meshes. Typically true if you have blending enabled for a mesh.

4.1.13 Enable triple buffering

By using triple buffering, the dependencies between application, graphics driver and GPU can be further relaxed. Also frame rates are no longer limited to integer divisors of the display refresh rate. Uniformly moving objects can start to jitter if triple buffering is used. Use eglSwapInterval() to get a constant redraw rate.

4.2 Optimizing Shader Programs

The goal of shader optimizations is to increase its throughput (vertices/sec, fragments/sec) without impacting visual appearance.

4.2.1 Use dedicated shader programs instead of universal shaders

One shader program can contain different functionalities that can be enabled/disabled through uniforms. An example would be fog: You can have a uniform that enables/disables the fog calculation in the shader or you can use two different shader programs, one with fog calculation and one without. In almost all cases, dedicated shader program yield the better performance.
4.2.2 Avoid if-else constructs in the shader

4.2.3 Only perform calculations in the shader program that need to be done at run time
Calculations in a shader program should only be done if they can’t be done outside the shader. Examples are:

- Calculations that do only require uniforms: Perform these calculations upfront in your application.
- Dedicated instructions to mirror, i.e. reflect, a model. Incorporate the reflection into the MVP matrix.

4.2.4 Carefully design your lighting

- Use per vertex lighting instead of per fragment lighting.
- Use as less lighting sources as possible (usually one lighting source is ok).
- Use simplified lighting calculations instead of just re-implementing the fixed function pipeline lighting.
- Pre-calculate your lighting effects by putting them into textures.

4.2.5 Minimize the number of (temporary) variables

4.2.6 Don’t use too many assigns

```cpp
float x = 0;
x = in;
var = 3 * x;
Better:
var = 3 + in;
```

4.2.7 Don’t allocate variables across a large code range, i.e. minimize their lifetime

```cpp
float x = in + 1;
//do many other calculations
out = 3 * x;
```

4.2.8 Make sure attribute precision in vertex shader and precision of vertex data in memory is identical
The precision of the attributes in the vertex shader must be identical to the precision of the vertex data in memory. Otherwise the graphics driver has to convert the vertex data every time the mesh is drawn. A common mistake is to have float data in memory and use mediump precision for attributes in the vertex shader. It is recommended to use float data in memory and a highp precision for attributes in the vertex shader.

4.2.9 Use mediump precision instead of highp (especially fragment shader)
The shader performs most of the arithmetic calculations much faster in mediump than in highp precision. When using mediump precision for vertex attributes, make sure the data in memory is stored in half-float format as well. Otherwise the driver has to convert the
precision everytime the object is drawn. Medium precision should be the default precision in the fragment shader and for varyings.

### 4.2.10 Perform vector based operations if possible

```glsl
vec2 x1 = (in1 + in1) / 2;
vec2 x2 = (in2 + in2) / 2;
```

Better:

```glsl
vec4 x.xy = in1;
x.zw = in2;
x = (x + x) * 0.5;
```

### 4.2.11 Use multiplication instead of division

### 4.3 Optimizing Textures

The goal of texture optimizations is to minimize the memory bandwidth required to fetch the texture data and to increase the texture cache hit rate. MB86R1X ‘Emerald-X’ has a 32kB texture cache. It can hold 8192 texels in 32 bit color format or 16384 texels in 16 bit color format.

#### 4.3.1 Make your texture images as small as possible

Texture images never need to be bigger than the maximum pixel area they cover in the framebuffer. For example if a textured mesh covers a rectangle area of 64x64 pixel in the framebuffer, the texture image should not exceed this size. Otherwise texture cache hit rate, memory consumption and texture image appearance (unless you use mip-mapping) is worse.

If possible, i.e. quality of visual appearance is ok, make the texture image even smaller than the pixel area covered in the framebuffer. This is usually possible for intensity, luminance and alpha Textures.

#### 4.3.2 Use a color format that requires less bpp (Bits Per Pixel)

By using a 16 bit color format instead of a 32 bit color format, the memory bandwidth required to fetch the texture data is halved and twice as many texels fit into the texture cache. For alpha and luminance use the 8 bpp format.

#### 4.3.3 Use Mipmaps

If an object changes its distance to the camera during rendering, mipmaps should be generated for the texture(s) applied to this object. In most cases mipmaps help to increase graphics performance as the right texture size is automatically chosen by the texture unit. Performance can only degrade if the texture unit has to “jump” through the different mipmap levels. In this situation the texture cache hit rate is reduced and performance slows done. Use a non-mipmaped texture in such situations.

#### 4.3.4 Draw objects sorted by texture

This way the texture cache isn’t flushed between objects.
4.3.5 Check if texturing is really needed
In some situations texturing of a mesh can be replaced by setting the color through a uniform in the fragment shader (e.g. paint of a car). Do also think about only getting the alpha or intensity value from the texture.

4.3.6 Use per quad texel access for supersampled scenes (compiler option)