Fujitsu Semiconductor Europe GmbH

Emerald Graphics Driver

Optimizing Graphics Applications

Application Note

Project-ID 03-00003-001
Revision 00.40
Date 2011-12-07

For Public Use

The information in this document is subject to change without notice.

Published by

Fujitsu Semiconductor Europe GmbH (FSEU)
Pittlerstrasse 47
63225 Langen
GERMANY

Attention please!

The information herein is given to describe certain components or functionalities and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved. We hereby disclaim any all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Revision History

Revision	Date	Changes (incl. section)
00.30	2011-11-04	Added information about GPU, Pixel Engine load measurement feature and Pixel Engine access to OpenGL buffers.
00.31	2011-11-07	Fixed formatting
00.40	2011-12-07	Added information about display controller layer buffering, Shader compiler –M5 option and mesh tessellation to improve texture performance.

Warranty and Disclaimer

The use of the deliverables (e.g. software, application examples, target boards, evaluation boards, starter kits, schematics, engineering samples of IC's etc.) is subject to the conditions of Fujitsu Semiconductor Europe GmbH ("FSEU") as set out in (i) the terms of the License Agreement and/or the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials.
Please note that the deliverables are intended for and must only be used for reference in an evaluation laboratory environment.
The software deliverables are provided on an as-is basis without charge and are subject to alterations. It is the user's obligation to fully test the software in its environment and to ensure proper functionality, qualification and compliance with component specifications.
Regarding hardware deliverables, FSEU warrants that they 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.
Should a hardware deliverable turn out to be defect, FSEU's entire liability and the customer's exclusive remedy shall be, at FSEU´s sole discretion, either return of the purchase price and the license fee, or replacement of the hardware deliverable or parts thereof, if the deliverable is returned to FSEU 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 FSEU, or abuse or misapplication attributable to the customer or any other third party not relating to FSEU or to unauthorised decompiling and/or reverse engineering and/or disassembling.
FSEU does not warrant that the deliverables do not infringe any third party intellectual property right (IPR). In the event that the deliverables infringe a third party IPR it is the sole responsibility of the customer to obtain necessary licenses to continue the usage of the deliverable.
In the event the software deliverables include the use of open source components, the provisions of the governing open source license agreement shall apply with respect to such software deliverables.
To the maximum extent permitted by applicable law FSEU disclaims all other warranties, whether express or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the deliverables are not designated.
To the maximum extent permitted by applicable law, FSEU's liability is restricted to intention and gross negligence. FSEU is not liable for consequential damages.
Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect.

The contents of this document are subject to change without a prior notice, thus contact FSEU about the latest one.

Table of Contents

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).

1.1 Scope

Only graphics driver aspects are covered by this application note.

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) as small as possible. Usually 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 Optimize display controller 0/1 layer assignment to avoid pixel distortion

If you face pixel distortion on the display caused by a memory bandwidth shortage, i.e. the display controller can't read the pixel data fast enough (function mmlGdcDispGetAttribute() can be used to check this) consider the following:

- Do only use odd or even display controller layers (e.g. L0, L2, L4, L6 or L1, L3, L5, L7) if possible. Reason: The layer pairs L0/L1, L2/L3, L4/L5 and L6/L7 share the same pixel prefetch buffer. If only one layer of a layer pair is enabled, it can use the whole prefetch buffer. Otherwise the prefetch buffer size available to a layer is halved. This results in a reduced memory latency tolerance for these layers. In case you need more than 4 layers, combine layers with a 16 bpp format or layers that are located in non-overlapping display areas.
- If pixels at the start of a display line show distortion, it can be caused by a non-optimal display timing: The time between HSYNC and first active pixel is to short (the display controller starts to fetch the pixels of a line just after the HSYNC). To solve this problem, modify the display timing to increase the time between HSYNC and first active pixel.

2.3 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. As 2D and 3D graphics operations run in parallel, you might see a performance increase just by loading the 2D and 3D pipeline more evenly.

2.4 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 mmlGdcConfigSetAttribute() 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: mmldGdcDispSyncWindowSwap(), 3D: elgSwapInterval()). This can be used to get a constant frame rate and/or better distribution of graphics performance among threads.

2.5 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 mmlGdcPeFinish() or 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). Do also consider your buffer requirements (triple vs. double buffering) to avoid waits caused by unavailable buffer.

2.6 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!

2.7 Check GPU and Pixel Engine load

The graphics driver offers hooks to query the GPU and Pixel Engine load. This can be used to find out if the graphics performance of an application is limited by the GPU or Pixel Engine (load would be 100%) or the application itself is causing the performance limitation (e.g. CPU load to high, i.e. graphics calls can't be generated fast enough; application contains waits). Under Linux you can access this information as follows:

```
cat /proc/driver/emeralddisplay/card
```

Note that the load is measured between 2 successive calls of "cat /proc/driver/emeralddisplay/card", so the first print-out must be ignored.

It is helpful to combine the GPU/Pixel Engine load measurement with the CPU load measurement.

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 defines 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 artifacts.
In some cases also reduced customized 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 realized 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 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()
- glStencilXXX()
- glTexImage2D()
- glTexParameterXX()
- glUniformXXX()
- 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. Note that the Pixel Engine can access the OpenGL color, stencil and depth buffers. So you can also save/restore the color, stencil and depth buffer of a static background image. See the Surface API for the OpenGL driver for details.

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:

```
/* 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 if possible.

4.2.5 Minimize the number of (temporary) variables

4.2.6 Don‘t use too many assigns

```
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

```
float x = in + 1;
//do many other calulations
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. Mediump precision should be the default precision in the fragment shader and for varyings.

4.2.10 Perform vector based operations if possible

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

Better:

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

4.2.11 Use automatic shader load balancing (Shader Compiler –M5 option)

By using the Shader Compiler option '-M5' you can enable the automatic load balancing between vertex and fragment shader programs in the shader core. This is possible as MB86R1X 'Emerald-X' features a Unified Shader Core that processes both, vertex and fragment shader programs. By enabling the automatic load balancing, shader processing performance is automatically distributed to the vertex shader and fragment shader program as needed. Following are examples to illustrate this:

- Large triangles need a lot of fragment shader processing but little vertex shader processing
- Backfacing triangles do only need vertex shader processing and no fragment shader processing (if backface culling is enabled).

Please note that the '-M5' command line option is not the default option as some devices supported by the Shader Compiler don't support this feature. The current version of the Shader Compiler also doesn't list this option in the help text (but it is documented in the Shader Compiler User Manual), but all versions of the Shader Compiler do support the '-M5' option.

4.2.12 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.
Keep in mind that mipmaps can only be generated if the width and height of a texture is a power of two.

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 Tessellate large textured triangles

If large textures (>= 512x512 pixels) are mapped on large triangles (>= 128x128 pixels), e.g. for billboarding, the rendering performance can increase by tessellating the large triangles. Due to the additional vertices, performance can also decrease! Therefore it is very important to measure the rendering performance of both variants (with and without tessellation).

References

No/Name	Title	Source
1	Emerald Graphics Driver User Manual Version 0.09	Emerald_GraphicsDriver_UserManual.pdf

Abbreviations / Terminology

V	
VRAM	**V**ideo **R**andom **A**ccess **M**emory. Physical continuous memory area set aside in DDRAM to be used/managed by the graphics driver. Used to hold graphics data for Video Capture, Display, 2D Engine and 3D Engine.