Technical Brief

Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL

August 2005
02011-001_v01

DEVELOPMENT

Document Change History

Version	Date	Responsible	Reason for Change
01	00/00/00		Initial release
02	08/17/05	Ikrima Elhassan	Added Chris, Mark, and Eric's suggestions
03	08/17/05	Ikrima Elhassan	Added perf #s and section about motherboards & chipsets

Preface

Bandwidth bottleneck is a common occurrence in many applications. This technical brief discusses how to achieve efficient bandwidth rates in your application, including transfers from and to the graphics processing unit (GPU).

Ikrima Elhassan
sdkfeedback@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

August 16, 2005

Achieving Efficient Bandwidth Rates

Graphics applications are often bandwidth bound. To make matters worse, non PCI-e video cards have asymmetric bandwidth rates, with slower readback rates than download rates.

Unfortunately, applications often require reading the frame buffer. For example, some applications write intermediary results to the frame buffer and use it as an input to additional rendering or computation passes.

This paper aims to discuss ways to achieve fast transfers by:

- Using an nForce3 or comparable AGP chipset
- Using pixel sizes that are multiples of 32 bits to avoid data padding.
- Storing 8-bit textures in a BGRA layout in system memory and use the **GL_BGRA** as the external format for textures to avoid swizzling.

Using a PBO to transfer data to and from the GPU. If possible, using multiple PBOs to implement an asynchronous readback scheme.

Motherboard System Configuration

The motherboard system configuration plays an important role in achieving fast texture downloads and readbacks. For both AGP 8x and PCIe machines, use an nForce3 or comparable AGP chipset to achieve higher transfer rates. Using a Geforce 6-series card or higher with these chipsets allows for higher speed readbacks than when using an nForce2 or comparable chipset.

Set the Correct Image Format

As a developer, you need to pay attention to the texture formats to ensure efficient transfer rates. Different texture formats affect performance differently.

Pixel Size

To begin with, make sure the pixel sizes are an integer multiple of 32-bits; if you fail to do this, the driver will perform data padding, which causes the transfer rate to slow down.

Pixel Format

For 8-bit textures, NVIDIA graphics cards are built to match the Microsoft GDI pixel layout, so make sure the pixel format in system memory is BGRA.

Why are these formats important? Because if the texture in system memory is laid out in RGBA, the driver has to swizzle the incoming pixels to BGRA, which slows down the transfer rate. For example, in the case of **`glTexImage2D(),`** the format argument specifies how to interpret the data that is laid out in memory (such as `GL_BGRA`, `GL_RGBA`, or `GL_RED`); the `internalformat` argument specifies how the graphics card internally stores the pixel data in terms of bits (`GL_RGB16`, `GL_RGBA8`, and `GL_R3_G3_B2`, to name a few). To make matters more confusing, OpenGL allows you to specify `GL_RGBA` as an internal format, but this is taken to mean `GL_RGBA8`. It is always best to explicitly specify the number of bits in the internal format. Refer to Table 1 to see the performance impact of using non-optimal texture formats. Note, this is not the case with 16-bit and 32-bit floating point formats.

Here are code snippets showing what to avoid and how to achieve fast transfer rates:

```
//These calls will cause a slow down because of
driver swizzling
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA8 ,
img_width, img_height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
img_data);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA ,
img_width, img_height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
img_data);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0,
GL_FLOAT_RGBA16_NV , img_width, img_height, 0, GL_BGRA,
GL_HALF_FLOAT_NV, img_data);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA_NV,
img_width, img_height, 0, GL_BGRA, GL_FLOAT_NV, img_data);

//These calls would not require unnecessary
swizzling
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA8 ,
img_width, img_height, 0, GL_BGRA, GL_UNSIGNED_BYTE,
img_data);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA ,
img_width, img_height, 0, GL_BGRA, GL_UNSIGNED_BYTE,
img_data);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0,
GL_FLOAT_RGBA16_NV , img_width, img_height, 0, GL_RGBA,
GL_HALF_FLOAT_NV, img_data);
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA_NV,
img_width, img_height, 0, GL_RGBA, GL_FLOAT_NV, img_data);
```

Table 1. Performance Impact of Using Non-Optimal Texture Formats using an AMD64 Athlon 3500+, 1 GB of RAM, using an nForce4 chipset. A 1024x1024 texture is used for measuring transfer

Readback	**FX8RGBA**	**FX8BGRA**	**FP16RGBA**	**FP16BGRA**	**FP32RGBA**	**FP32BGRA**
GeForce 6800 Ultra	768 MB/s	790 MB/s	602 MB/s	650 MB/s	568 MB/s	581 MB/s
GeForce 7800 GT	770 MB/s	718 MB/s	630 MB/s	653 MB/s	640 MB/s	677 MB/s
Quadro FX 4400	789 MB/s	824 MB/s	787 MB/s	827 MB/s	710 MB/s	738 MB/s
Quadro FX 4500	745 MB/s	805 MB/s	760 MB/s	806 MB/s	770 MB/s	810 MB/s
Download						
GeForce 6800 Ultra	480 MB/s	1596 MB/s	1583 MB/s	490 MB/s	1734 MB/s	480 MB/s
GeForce 7800 GT	483 MB/s	2238 MB/s	1987 MB/s	527 MB/s	2367 MB/s	525 MB/s
Quadro FX 4400	471 MB/s	2167 MB/s	2157 MB/s	503 MB/s	2237 MB/s	463 MB/s
Quadro FX 4500	490 MB/s	2605 MB/s	2613 MB/s	493 MB/s	2689 MB/s	527 MB/s

Use the Pixel Buffer Object Extension

To achieve fast transfers to and from the graphics card, applications should use the Pixel Buffer Object (PBO) extension. Conceptually, a PBOs is simply an array of bytes in memory.

- To achieve a fast readback, bind the buffer object to GL_PIXEL_PACK_ BUFFER using **`glBindBufferARB()`** After a buffer is bound, **`glReadPixels()`** will pack (write) data into the Pixel Buffer Object.
- To download data to the GPU, bind the buffer to the **GL_PIXEL_UNPACK_ BUFFER** target using **`glBindBufferARB()`**. The **`glTexImage2D()`** command will then unpack (read) their data from the buffer object.
- To modify the data in the PBO, use **`glMapBuffer()`** to retrieve a pointer to the PBO's data. Once data modification is completed, the application must issue a **`glUnmapBuffer()`**. All updates to the buffer object must be done between these two calls.

These PBOs can improve performance because they allow the driver to streamline reading and writing to and from video memory. For example, when streaming textures, using PBOs and **`glMapBuffer()`** and **`glUnmapBuffer()`** usually eliminates an expensive data copy that is usually required for downloading a texture to the GPU.

Implement an Asynchronous Readback

Problem

OpenGL makes it difficult to pipeline readback of multiple images. For example, if the application requests readback data using **`glReadPixels`**, the driver often has to send the hardware a readback command and wait for all the data to return before it can let the application proceed.

This stall prevents the application from processing the readback data while kicking off another glReadPixels(). Moreover, stalls can occur because any impending commands to the frame buffer must be completed before readback can begin. This is essentially the same as a **`glFinish()`** before every **`glReadPixels()`**.

Solutions

If, however, you use PBOs, the application can work around these stalls and perform asynchronous readback of the frame buffer. When you use PBOs, **`glReadPixels()`**returns asynchronously and doesn't wait for the data to return from the GPU. Instead, the driver ensures that all the data to be read back is ready when the application issues the **`glMapBuffer()`** command.

By using multiple PBOs, such as the two explained below, the application can asynchronously read back data.

Split the Frame Buffer into Multiple Portions

One way of asynchronously reading back data is to split the frame buffer into multiple portions and map those into different PBOs. For example, if we split the frame buffer in half, the application could issue **`glReadPixels()`** from the top half of the frame buffer into PBO1, and issue **`glReadPixels()`** from the bottom half of the frame buffer into PBO2. Because **`glReadPixels()`** returns asynchronously when using PBOs, both transfers are kicked off simultaneously. Then, the application can map the top portion of the frame into PBO1, causing any outstanding Direct Memory Access (DMA) transfers to finish.

The benefit is that the application is not being stalled by DMA transfers into the bottom half of the frame. This means the application can perform calculations on the readback data from the first half of the frame buffer while data is still being readback from the second portion of the frame. This allows the application to pipeline CPU processing of readback data and continue to read back data from the frame buffer. Be aware, however, this method still requires rendering of the image to complete before it is read back. In other words, there is always an implicit **`glFinish()`** before each **`glReadPixels()`**.

Map Different Frames to Different PBOs

Another way of asynchronously reading back data requires mapping different frames to different PBOs. For example, using two PBOs, the application calls **`glReadPixels()`** into PBO1 at frame *n*. At frame *n+1,* the application calls **`glReadPixels()`** into PBO2 and then processes the data in PBO1. Ideally, enough time should pass between the readback calls to allow the first to complete to before the CPU begins processing it.

By alternating between PBO1 and PBO2 on every frame, asynchronous readback can be achieved. Moreover, applications can wait two or more frames.

Conclusion

To achieve efficient bandwidth rates, applications should implement fast download and readback paths. Using the techniques outlined in this technical brief should improve performance of data transfers.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com