User Guide

TexturePerformancePBO Demo

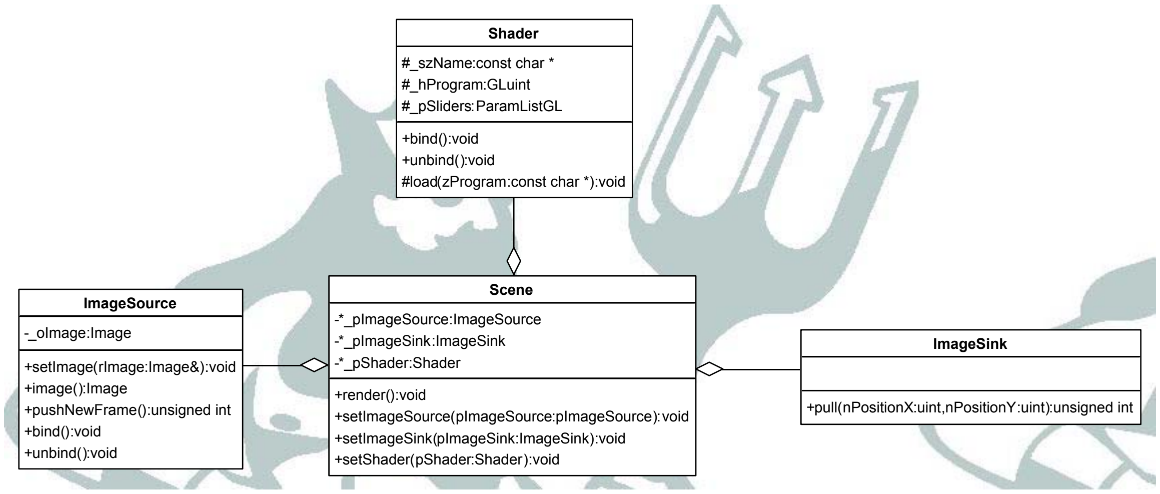
The TexturePerformancePBO Demo serves two purposes:

1. It allows developers to experiment with various combinations of texture transfer methods for texture upload (to the GPU) and readback. Performance implications of the various combinations can be explored interactively.
2. The demo code can be used as an example implementation on how to use the EXT_pixel_buffer_object extension in applications to perform fast DMA transfers of pixel data to and from the GPU. The simple structure of the demo allows developers to easily extend the application to add their own performance experiments.

Class Design

Core Functionality

At the core of the TexturePerformancePBO demo three classes work together to download images to the GPU, render/process these images, and read the processed data back to system memory: Scene, ImageSource, and ImageSink.

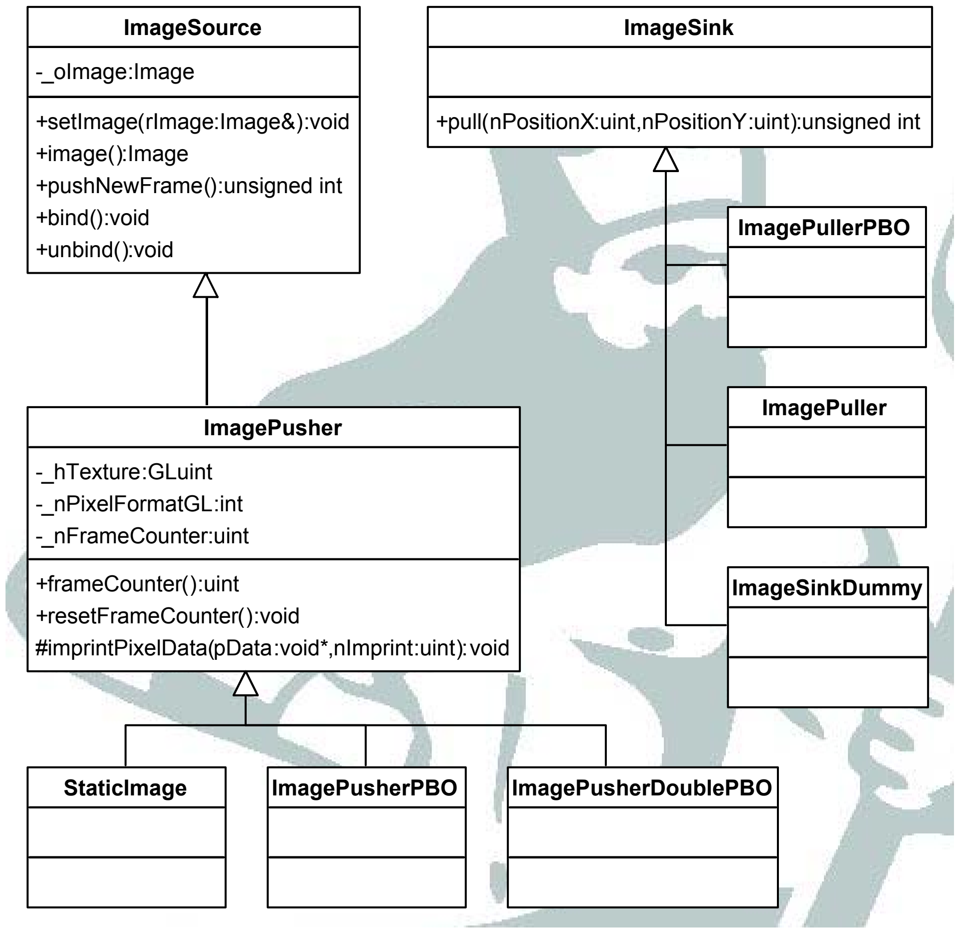

Shader
#_szName:const char *
#_hProgram:GLuint
#_pSliders:ParamListGL
+bind():void
+unbind():void
#load(zProgram:const char *):void
Scene
-*_pImageSource:ImageSource
-*_pImageSink:ImageSink
-*_pShader:Shader
+render():void
+setImageSource(pImageSource:pImageSource):void
+setImageSink(pImageSink:ImageSink):void
+setShader(pShader:Shader):void
ImageSource
-_oImage:Image
+setImage(rImage:Image&):void
+image():Image
+pushNewFrame():unsigned int
+bind():void
+unbind():void
ImageSink
+pull(nPositionX:uint,nPositionY:uint):unsigned int

ImageSource provides the texture to be rendered. Its bind() function binds the input image to the current OpenGL texture unit. Before calling bind() the pushNewFrame() method should be called to make sure a texture is actually available. pushNewFrame() returns the number of bytes that were transferred between system memory and the GPU.

Scene renders a screen-aligned quad to the frame buffer. The render() method performs a pushNewFrame() on the Scene's ImageSource and binds the new texture. Scene uses its Shader to render the texture to the frame buffer. A Shader might for example perform gamma correction, enhance contrast, etc. After rendering is complete the Scene calls pull() on its ImageSink.

ImageSink reads data from the frame buffer back to system memory. The pull() method returns the number of bytes transferred from the GPU to system memory.

ImageSource and ImageSink both are abstract base classes. The various classes derived from ImageSource and ImageSink implement different transfer mechanisms.



ImageSource has four concrete implementations:

1. ImagePusher which uses normal glTextureSubImage() to upload texture data to the GPU.
2. StaticImage which creates a texture and uploads it only once. Subsequent calls to pushNewFrame() will not cause any new data to be transferred to the GPU.
3. ImagePusherPBO which uses the EXT_pixel_buffer_object to repeatedly transfer new texture content to the GPU (dynamic texture usage and write only modification).
4. ImagePusherDoublePBO uses two separate pixel-buffer objects, one to kick of a new transfer and one to be used for rendering. This should further decouple data transfer and rendering and yield better performance than just a single PBO.

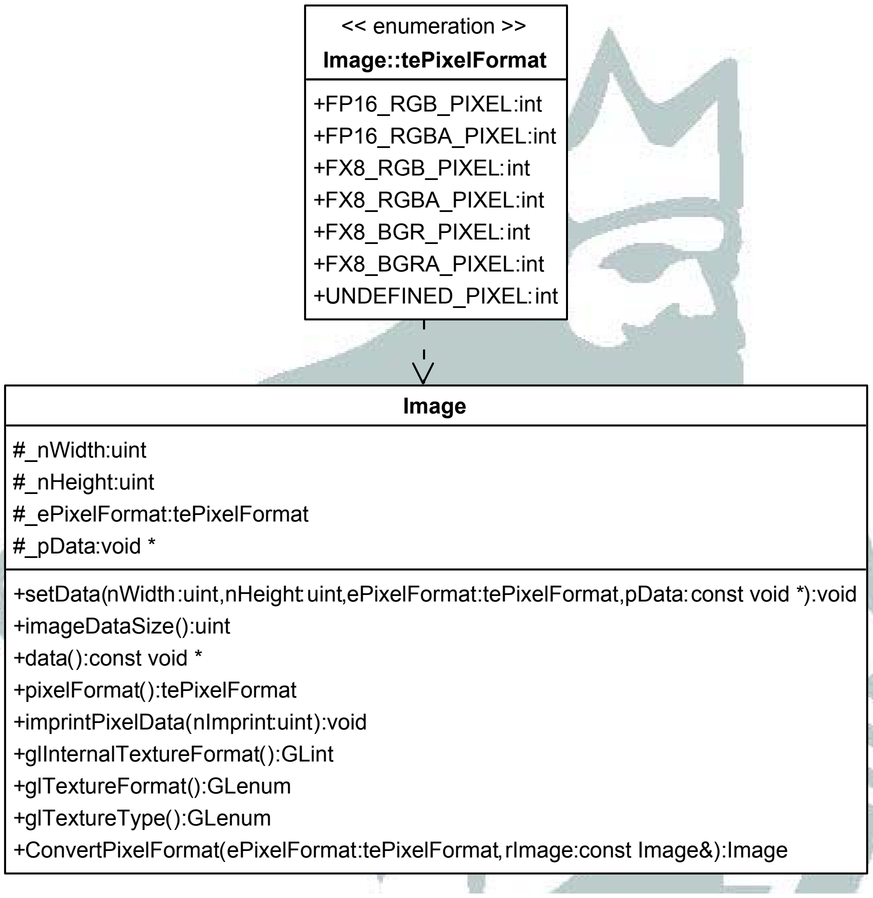
ImageSink has three concrete implementations:

1. ImageSinkDummy which doesn't read back any data from the GPU.

2. ImagePuller which uses glReadPixels() in a straight forward manner to transfer the frame-buffer content back from the GPU into system memory.
3. ImagePullerPBO which uses a pixel-buffer object to read frame-buffer data asynchronously from the GPU.

Supported Image Formats

Image and texture formats can have a massive impact on the performance of image transfer to- and from the GPU. The reason is that in case of a format mismatch OpenGL needs to perform a time-consuming format conversion.
In order for a developer to be able to experiment with the different combinations it is necessary for this demo to support at least the commonly used formats. The external formats (i.e. formats of images stored in system memory) supported are 16bit floating point RGB and RGBA, 8 bit unsigned char RGB, RGBA, BGR, and BGRA. Texture formats supported (i.e. format of how images are stored on the GPU) are 16 bit floating point RGB and RGBA, 8 bit unsigned integer RGB, and RGBA.
Since readback is only performed on an 8 bit RGBA frame-buffer the different formats are only supported for texture upload to the GPU. The two classes involved are the ImageSource and Image.



<< enumeration >>
Image::tePixelFormat
+FP16_RGB_PIXEL:int
+FP16_RGBA_PIXEL:int
+FX8_RGB_PIXEL:int
+FX8_RGBA_PIXEL:int
+FX8_BGR_PIXEL:int
+FX8_BGRA_PIXEL:int
+UNDEFINED_PIXEL:int
Image
#_nWidth:uint
#_nHeight:uint
#_ePixelFormat:tePixelFormat
#_pData:void *
+setData(nWidth:uint,nHeight:uint,ePixelFormat:tePixelFormat,pData:const void *):void
+imageDataSize():uint
+data():const void *
+pixelFormat():tePixelFormat
+imprintPixelData(nImprint:uint):void
+glInternalTextureFormat():GLint
+glTextureFormat():GLenum
+glTextureType():GLenum
+ConvertPixelFormat(ePixelFormat:tePixelFormat, rImage:const Image&):Image

Image knows about its image format. In order for setData() to work correctly the right image format enumerator needs to be specified. Based on this information glTextureFormat() and glTextureType() return the correct OpenGL tokens needed for texture creation using glTexImage(). glInternalTextureFormat() returns the internal texture format best suited for this image's data format.
ImagePusher itself contains an option to define an internal image format to be used for GPU upload. If the ImagePusher finds that the user set a specific internal format than this format it used for the texture upload. Otherwise ImagePusher uses the format returned by Image::glInternalTextureFormat().
For easy creation of various image formats Image also contains a static conversion method ConvertPixelFormat. This method takes an Image and a target format and returns a copy of this image in the new format. In the demo app this method is used to create copies of the sample image in all the different storage formats supported by Image. When the user changes to a new format all the different incarnations of ImageSources are set up so that they use this image with the new format.
An other feature of the image class is the imprintPixelData() method. This method imprints a binary time stamp (nImprint) in the lower left corner of the image. Note: This happens in system memory. The ImagePushers increments this time stamp before each new upload of an image. When the application runs and images continuously get uploaded and rendered this shows the progress and also proves that the images in fact get uploaded between each render pass.

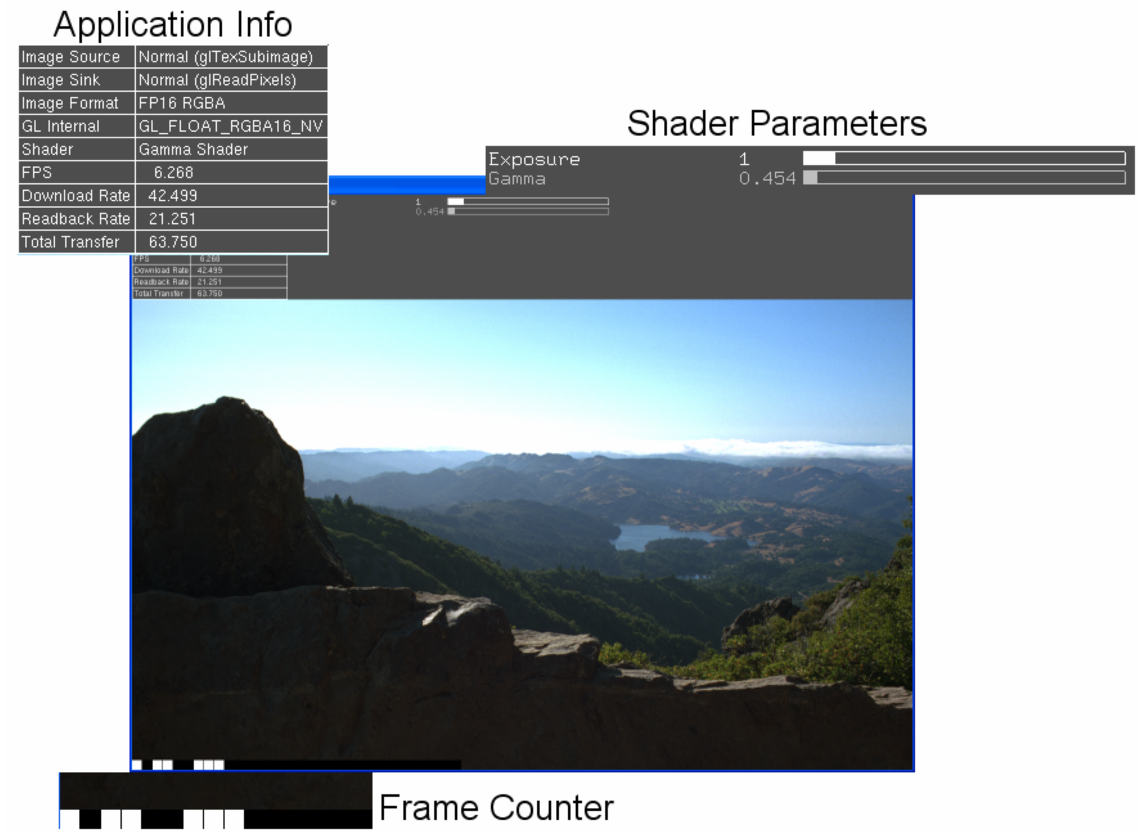
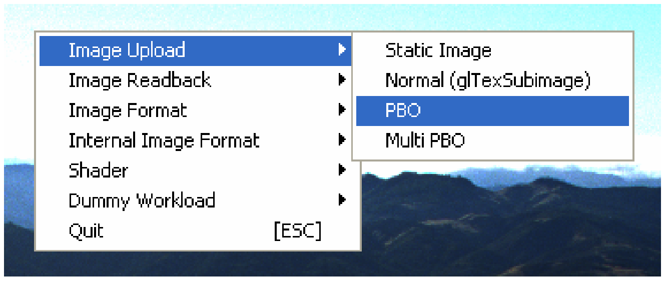


Image Source	Normal (glTexSubimage)
Image Sink	Normal (glReadPixels)
Image Format	FP16 RGBA
GL Internal	GL_FLOAT_RGBA16_NV
Shader	Gamma Shader
FPS	6.268
Download Rate	42.499
Readback Rate	21.251
Total Transfer	63.750

Exposure	1
Gamma	0.454

Demo App in Action

Here is a screen shot of the TexturePerformancePBO demo with the different screen elements magnified. The application info area shows all the options used for texture download, readback, shader, image format, internal texture format, and the current transfer rates upstream, downstream, and the sum.
The shader parameter area allows the user to interactively manipulate shading. To change a parameter the up- and down-arrow keys are used to cycle through the parameter list. The active parameter is displayed in white. The parameter value is changed using the left- and right-arrow keys.
In the right bottom corner the frames "time stamp" is displayed as a binary pattern. The time stamp is imprinted on the image in system memory before it is transferred to the GPU. When "Static Image" is selected as image source (which only uploads the texture to the GPU once) no time stamp shows up. In this case the download rate will also show up as zero since no data is transferred to the GPU.
Setting different image sources, sinks, etc. is accomplished via a GLUT menu. Right click brings up the menu.



One thing shown in the image above has not yet been mentioned: The "Dummy Workload". When using PBOs for read pixels ideally these data transfers are performed as DMA transfers by the memory controller and thus only burn a minimum amount of CPU cycles. Since the demo app is very light-weight in terms of CPU in the first place there is hardly any improvement visible when using asynchronous PBO read-backs instead of normal read-backs. What enabling the dummy workload does is to perform some dummy CPU stuff after issuing the readback call but before accessing the texture. Since the DMA transfer happens parallel to the CPU working the performance improves compared to the normal read-back that consumes the CPU for transfer and executes the dummy workload after finishing the read-back.

Interesting Experiments

The most interesting thing for most developers is most likely to find out what combination of internal and external format and what technique gives them maximum texture transfer speeds.
One of the general rules is that the pixel size should be an integer multiple of 32bits. For all other cases some kind of data-padding occurs slowing down transfer. The other issue is the order of the color channels. NVIDIA cards are build to match the Microsoft GDI pixel layout for the 8 bit per channel formats. This pixel format in fact is BGRA in system memory. When colors in RGBA format are transferred these need to be swizzled during transfer.

Maximum download speeds for 8 bit per channel images are achieved using the BGRA image format and RGBA as internal format. Switching through the different ImagePushers reveals that PBO is necessary for maximum transfer rates.

On lower end graphics cards it is also good to check, what the maximum render rate of the card is to make sure that image transfer is in fact the bottleneck and not the display of the image. Setting the image source to static image, image sink to dummy basically results in the GPU rendering the image as fast as possible (v-sync should be set to always off in the NVIDIA control panel). Switching the simple and the gamma shader should demonstrate that the shader has a pretty big impact on rendering performance. If the FPS (frames-per-second) is substantially higher than the maximum frame rate achieved with texture transfers it is clear that the shader will not become the bottleneck.

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