# Reading the Visible Frame Buffer to a Pixel Buffer Object

Antonio Díaz-Tula
Departamento de Ciencia de la Computación
Universidad de Oriente
Ave. Patricio Lumumba, 9500
Santiago de Cuba, Cuba
diaztula1@gmail.com

Miguel Castañeda-Garay
Departamento de Ciencia de la Computación
Universidad de Oriente
Ave. Patricio Lumumba, 9500
Santiago de Cuba, Cuba
mcgaray_cu@yahoo.es

Óscar Belmonte-Fernández
Departamento de Ingeniería y Cienca de los Computadores
Universitat Jaume I, Spain
Oscar.Belmonte@lsi.uji.es

## ABSTRACT

Under certain circumstances it is necessary to read the visible frame buffer from a display device for post-processing. Examples of such applications are systems where the user interacts through projection screens and cameras. Traditionally the frame buffer is copied to the RAM to perform the post-processing, which many times is parallelisable and compute-intensive, leading to high latency times associated with the transfer of the pixel's information over the bus and serial execution on CPU. But to the purpose of processing the frame buffer copy with the programmable hardware of the graphics processing unit (GPU), it needs to be copied to Video RAM instead of conventional RAM to avoid the latency time and exploit the parallel computational resources of the GPU. On this work we present an easy-to-use approach for reading the visible frame buffer to a Pixel Buffer Object, suitable for processing with the GPU through the CUDA architecture. A comparison with a method for transferring the visible frame buffer to the RAM is presented, as well as a practical application.

### Keywords

visible frame buffer, pixel buffer object, layered window, CUDA.

## 1. INTRODUCTION

Several approaches are based on post-processing the content of the visible frame buffer. For example, in high definition projection screen systems several computer vision techniques are based on post-processing the image that one or more projection screens are displaying, in order to perform comparisons with the images captured by one o more cameras, thus detecting the user actions like pointing to a certain region of the projected image [Bres03a], [Jay04a], [Mig09a].

If this processing is performed on CPU, there is a latency time associated with the transfer of the image being displayed by a projection device to the main memory (RAM), and the processing itself is done in serial execution.

With the rising and development of programmable video cards, new opportunities are now at close hand. These video cards have an internal video memory (VRAM) and a parallel architecture of several processors and enormous computational capabilities.



Examples of the software counterpart to exploit these features are the Shading Languages and CUDA.

The process of reading the frame buffer into VRAM is currently done by using a pixel buffer object (PBO), which is suitable for processing with CUDA since it can be mapped to be accessed from CUDA's kernels, either for reading and/or writing.

But for reading the "entire" visible frame buffer (not a window's specific) a strong background and programming skills in OpenGL and the target Operative System (OS) API is required.

On this work we present an approach encapsulated in a C++ class that abstracts all the complexity associated with the process of reading the visible frame buffer of a display device into VRAM using a PBO. The target OS is Microsoft Windows.

## 2. ACCESSING THE FRAMEBUFFER THROUGH OPENGL

To be able to use PBOs, the hardware-accelerated implementation of OpenGL must be launched.

Over Windows, the visible frame buffer is the Desktop. But the OS does not allow creating a hardware-accelerated Rendering Context (RC) associated with the Desktop and make it current.

A full-screen, top-level transparent window is needed, such window do support a hardware accelerated RC. A window's procedure is required,

as well as many calls and configuration setup for both OpenGL and OS API.

We encapsulate all the processing needed in a single C++ class, whose interface is detailed in Table 1.

| Method | Description |
|---|---|
| *w2GL* | Constructor: initializes the instance attributes, gets the width, height and color depth of the desktop; creates a transparent, top-level window that covers the entire desktop. |
| *~w2GL* | Destructor: releases all allocated resources and destroys the window. |
| *createGLContext* | Creates a device context for the window, assign it a native pixel format (hardware-accelerated) and creates the corresponding rendering context. |
| *makeCurrent* | Makes the previously created rendering context current. |
| *unmakeCurrent* | Makes the rendering context no longer current. |
| *readFrameBuffer* | Reads the frame buffer content to the pixel buffer object. |
| *getWidth* | Returns the window's width. |
| *getHeight* | Returns the window's height. |
| *getPBO* | Returns the pixel buffer object identifier (unsigned integer), which can be used for processing the frame buffer image either in GPU or CPU. |
| *displayChange* | Adjust the window's size, OpenGL current state and PBO memory to the new size and color depth of the screen. |

**Table 1. The interface of the class w2GL**

## 3. EXPERIMENTATION AND CURRENT APPLICATION

A method for reading the visible frame buffer to the RAM, based on a Memory Device Context and a Device Independent Bitmap (DIB) [MSD05], was implemented in order to perform some comparisons with our approach. This method includes a call to the function *CreateDIBSection* and *BitBlt* from the Windows's API. Results are shown in Table 2.

Although results show a better performance using Windows's DIB, not all devices support the *BitBlt* function, it is not suitable for processing in GPU and the pixel information is in BGR format.

| Screen resolution | Using DIB | Using *w2GL* |
|---|---|---|
| 1024x768 | 340 | 215 |
| 1280x1024 | 190 | 130 |

**Table 2. Performance results**

The class w2GL allows processing the visible frame buffer in both GPU and CPU. Full screen applications are captured, including those that use OpenGL and Direct3D. On the other hand, video rendering using hardware-supported overlays is not captured (a black region is read).

If a user of the class wants to draw something using OpenGL, it will need to create its own rendering context and make it current, since our class does not create a new thread, but is part of the one that created the object *w2GL*. This is a disadvantage since the context switching is expensive. Future extensions may create its own thread to overcome this difficulty.

The approach works for multiple screens. In the Dual View Mode, it will read the main screen.

The class is currently used in a parallel implementation of a local method for detecting non-stationary photometric perturbations in projected images [Mig09a] with CUDA, as one part of this method requires post-processing the visible frame buffer.

Results are good, as in the first experimentation a speedup of 2.8x was achieved, and ultimately up to 3.5x. Reading the visible frame buffer using our approach makes no degradation on the system performance.

## 4. REFERENCES


[Bre03a] Bresnahan, G. et al.: Building a large scale, high-resolution, tiled, rear projected, passive stereo display system based on commodity components. In: Stereoscopic Displays and Virtual Reality Systems X, SPIE Proc. Vol. 5006 (2003).

[Jay04a] JAYNES C., WEBB S., STEELE M.: Camera-based detection and removal of shadows from interactive multiprojector displays. IEEE Transactions on Visualization and Computer Graphics 10, 3 (2004).

[Mig09a] Castañeda-Garay, M., Belmonte-Fernández, O., Gil-Altaba, J., Pérez-Rosés, H., Un Método para la Detección en Tiempo Real de Perturbaciones Fotométricas en Imágenes Proyectadas, Congreso Español de Informática Gráfica CEIG'09, San Sebastian, Sept. 9-11 (2009), Páginas 239-242. The Eurographics Digital Library, http://diglib.eg.org.

[MSD05] Microsoft Development Network Library 2005.