REAL-TIME RENDERING OF MULTIPLE VIEWS USING STANDARD GRAPHICS HARDWARE

Chun-Wei Chan

Dec 15, 2005

Boston University

Department of Electrical and Computer Engineering

Technical report No. ECE-2005-05

BOSTON
UNIVERSITY

REAL-TIME RENDERING OF MULTIPLE VIEWS USING STANDARD GRAPHICS HARDWARE

Chun-Wei Chan

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

Dec 15, 2005

Technical report No. ECE-2005-05

Summary

The objective of this research is to improve the speed of processing multiview images before displaying them on automultiscopic display devices such as the “Synthagram” from Stereographics Corp. Displaying these images without hardware support takes a lot of time and thus is not in real time. In this report, we describe the process of pre-processing and rendering multiview images, investigate means of accomplishing both using general CPU (central processing unit) as well as GPU (graphics processing unit) under OpenGL, and compare performance of different approaches. Although none of the investigated methods is capable of real-time performance (i.e., at least 30 frames per second) for full-screen images, it is clear that GPU-based pre-processing and rendering under OpenGL outperforms CPU-based approach by at least an order of magnitude.

Contents

List of figures and tables

1. Introduction

Multiview autostereoscopic displays, also called automultiscopic displays, such as the “Synthagram” from Stereographics Corp., enable visualization of 3D scenes by rendering several views (9 in the case considered) captured from the said scene. In order to generate a displayable image capable of causing 3D sensation, the 9 original views are combined together in a process called “interzigging”; each subpixel (i.e., color component of a pixel – R, G or B) in the final image is extracted from one of the 9 source pictures. Since in the process of combining the 9 source images a single image of the same resolution is produced, it is clear that subsampling occurs which must be preceded by lowpass filtering in order to prevent aliasing.

Today, there are primarily two autostereo technologies: one based on the idea of microlens, also called lenticular, of which the “Synthagram” is an example, and the other based on the idea of parallax barrier, i.e., opaque sheet placed over a screen with narrow slits cut out. Both the lenticules and the slits serve as a light-directing layer that allows viewer’s eyes to see different sets of pixels on the screen.

The aliasing mentioned before may manifest itself as false colors at sharp object boundaries, colored noise throughout the image or even spurious patterns in textured areas. Given screen parameters, such as

- number of views,
- pixel pitch,
- lenticule or slit pitch and angle of orientation,

optimal anti-aliasing filters can be designed []. We assume that such filters are given and we are concerned only with applying them to the original images, and then with the subsampling and multiplexing. A block diagram describing the whole approach is shown in Fig. 1.

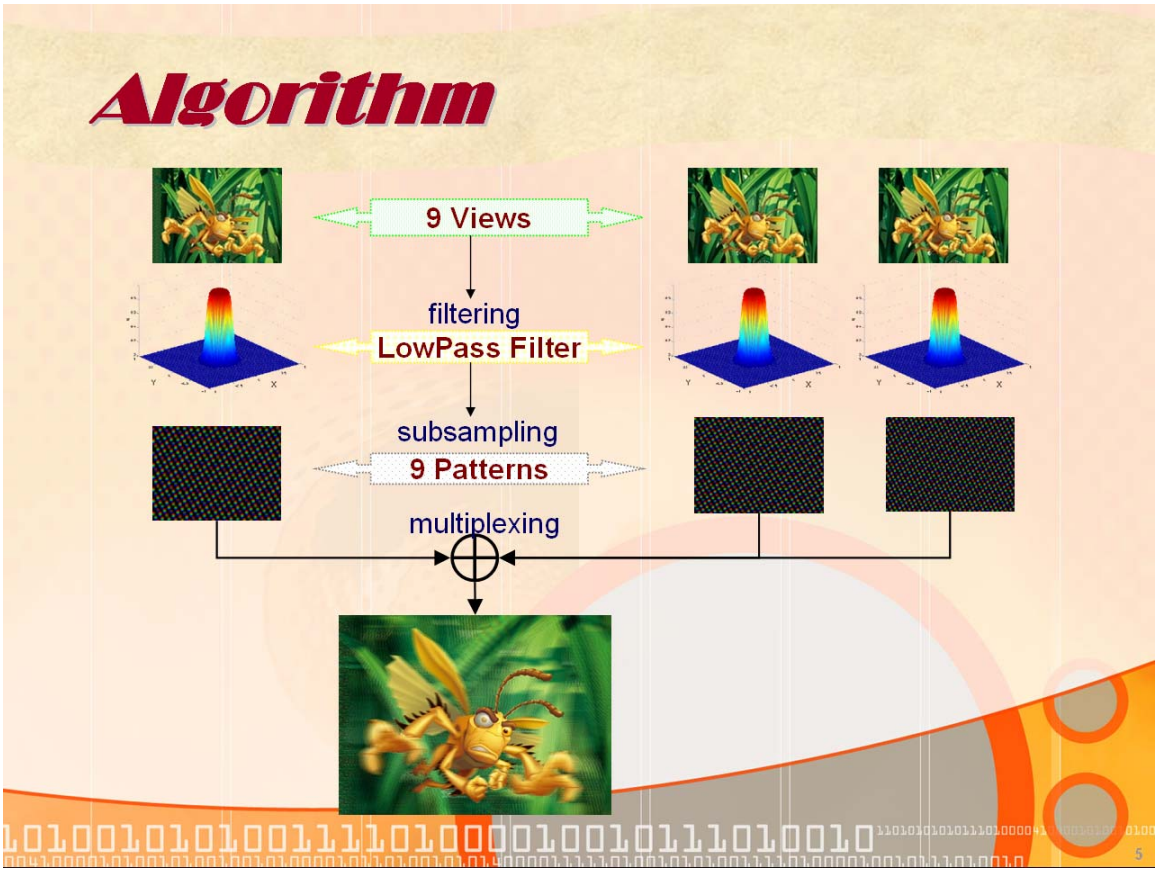

Figure 1 Block diagram of multiview rendering

Although the three steps of multiview rendering, namely, pre-filtering, subsampling and multiplexing can be implemented in CPU on any computer, for many applications the rendering speed is as important as the rendering quality. This research shows that the multiview rendering process can be significantly accelerated by modern graphics hardware, such as available in a typical computer graphics card. This report describes three implementations of the multiview rendering, one by a general CPU and two by a GPU, and presents a quantitative comparison of the methods' performance.

2. Computer Graphics Processing

First, we discuss how the computer graphics works. The basic hardware components and data flow involved in rendering multiview images using a CPU are shown in Fig. 2.

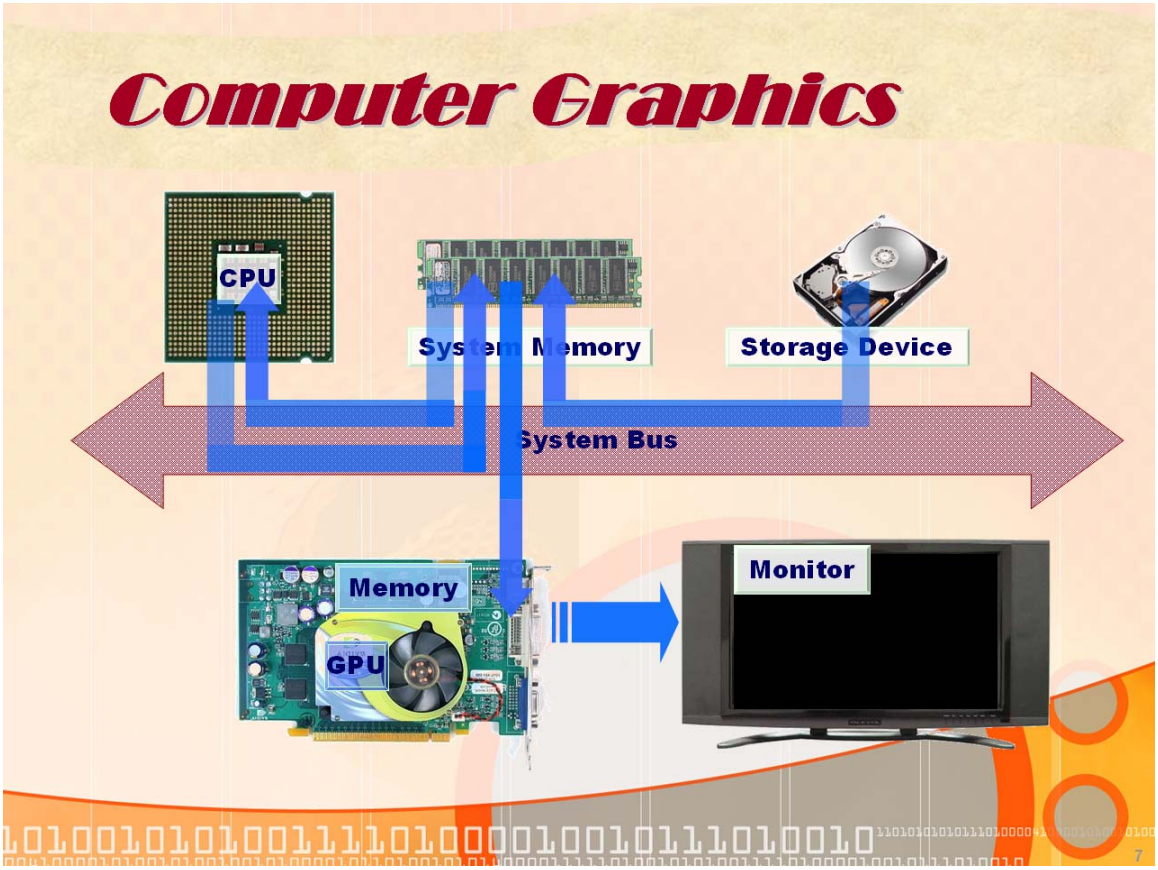

Figure 1 CPU-based multiview rendering

As Fig. 1 shows, in CPU-based multiview rendering images are first loaded from storage device into system memory. If the images need to be modified (e.g., anti-alias filtering), they are transferred to the CPU using the system bus, and after calculations are completed the new data are sent back to system memory. Since the goal is to show the resulting picture on the screen, the data are sent to the graphics card and then output to the display device. There are several steps here that can slow down the process. One is loading images from a hard drive, but in this project we assume the data have already been stored in the system memory. This is not unreasonable because in the envisaged 3D applications

images are captured by 9 cameras, that are likely to save the data directly in the system memory at a fixed memory address. The images can be accessed directly from systems memory without loading them from a storage device again. The second noticeable slowdown is outputting the data to a graphics card, and displaying the images on a screen. Here the bottleneck is in the data passing through the system bus to the graphics card. This problem can be only solved by faster hardware. The third and most important slowdown are calculations performed by the CPU. Assume that each of the 9 images is of n1 x n2 dimension, and that the anti-aliasing filters are of f1 x f2 dimension. The complexity is of the order O(n1xn2xf1xf2), because the number of, e.g., multiplications, needed is:

convolution	n1 x n2 x f1 x f2
3 color components	n1 x n2 x f1 x f2 x 3
9 images	n1 x n2 x f1 x f2 x 3 x 9

For example:

n1 = 1024, n2 = 1024, f1 = 7, f2 = 7

n1 x n2 x f1 x f2 x 3 x 9 = 1,387,266,048 multiplications

Even if the image size is small, for example 256x256, it takes a long time to complete these calculations. We have implemented a CPU-based multiview rendering and measured the speed of processing images of various sizes. The experimentally-measured number of processed frames per second are shown in Table 1. Clearly, the achieved rendering rate is from being in real time (about 30 frames/sec and above) and we need to consider another approach.

An alternative to CPU-based processing is GPU-based processing bypassing the CPU and performing all the calculations directly on a graphics card. In the

GPU mode, calculations are all done by a GPU which is the core of a graphics card and is all pipelined. The CPU performs only some data and instruction management. Moving this job from the CPU to a GPU is beneficial because the pipeline architecture is a better fit to image processing despite GPU's lower clock rate than that of a typical CPU. Another reason is the fact that while the performance increase of CPUs changes rather slowly, the performance of graphics cards is changing dramatically every year. Below, we show in Fig. 3 a typical data flow and hardware involved in GPU-based multiview rendering.

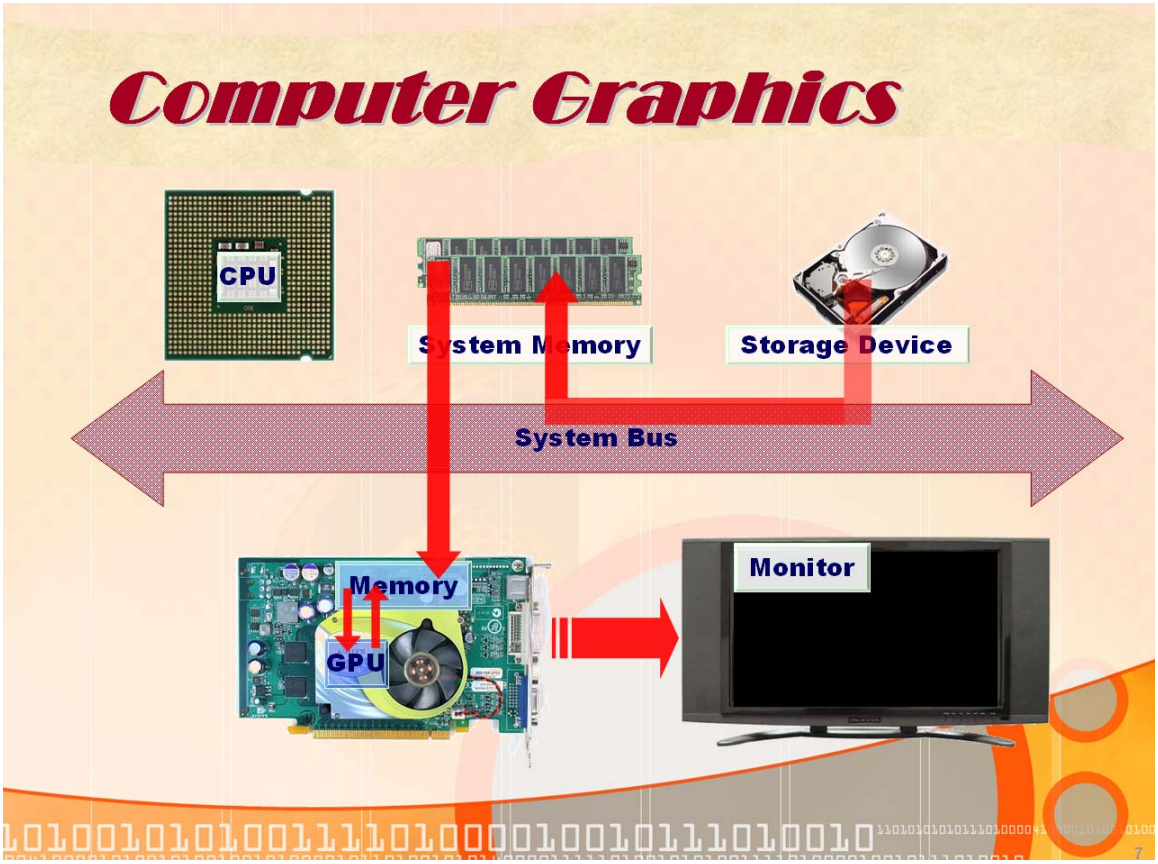

Figure 3 GPU-based multiview rendering

Below we investigate two approaches to the use of GPU in multiview rendering, one based on the OpenGL language and one based on the Shader language.

3. OpenGL

In order to maximally exploit graphics hardware capabilities with the lowest programming effort, typically either OpenGL or DirectX programming language is used. We use OpenGL in this project for a number of reasons listed below.

a. OpenGL

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

b. Industry standard

An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard.

c. Stable

OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes. Backward compatibility requirements ensure that existing applications do not become obsolete.

d. Reliable and portable

All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system.

e. **Evolving**
   Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. In this way, innovations appear in the API in a timely fashion, letting application developers and hardware vendors incorporate new features into their normal product release cycles.

f. **Scalable**
   OpenGL API-based applications can run on systems ranging from consumer electronics to PCs, workstations, and supercomputers. As a result, applications can scale to any class of machine that the developer chooses to target.

g. **Easy to use**
   OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

h. **Well-documented**
   Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain.

For our project, it is very important that convolution is a simple function call under OpenGL; only a filter must be specified. The problem in our case, however, is that we do not need to filter one image only, but we need filter 9 images at a time and then combine them together though subsampling and multiplexing. In other words, while the image has been passed through a filter, the filtered image should be sent back and stored in memory. This step makes the processing slow down. The reason is that the card we tested does not allow a user to store so much data in its memory. Thus, the performance of this

approach depends strictly on the amount of memory a graphics card possesses. As can be seen in Table 1, there is a significant performance difference between high-end (Wildcat II 5110) and entry-level (Quadro FX500) cards. However, the performance of GPU multiview rendering under OpenGL is faster than that of CPU-based rendering by the factor of about 5-10. A further speed-up may be possible if one can accelerate storing the filtered image data before the final multiplexing.

4. Shader Language

Another approach to using the GPU for multiview rendering is implement the method in a language called “shader language”. This language has been designed for programming GPUs, just like C is for CPUs. Before starting programming a GPU lets analyze how a GPU works. The discussion below is in view of Fig. 4.

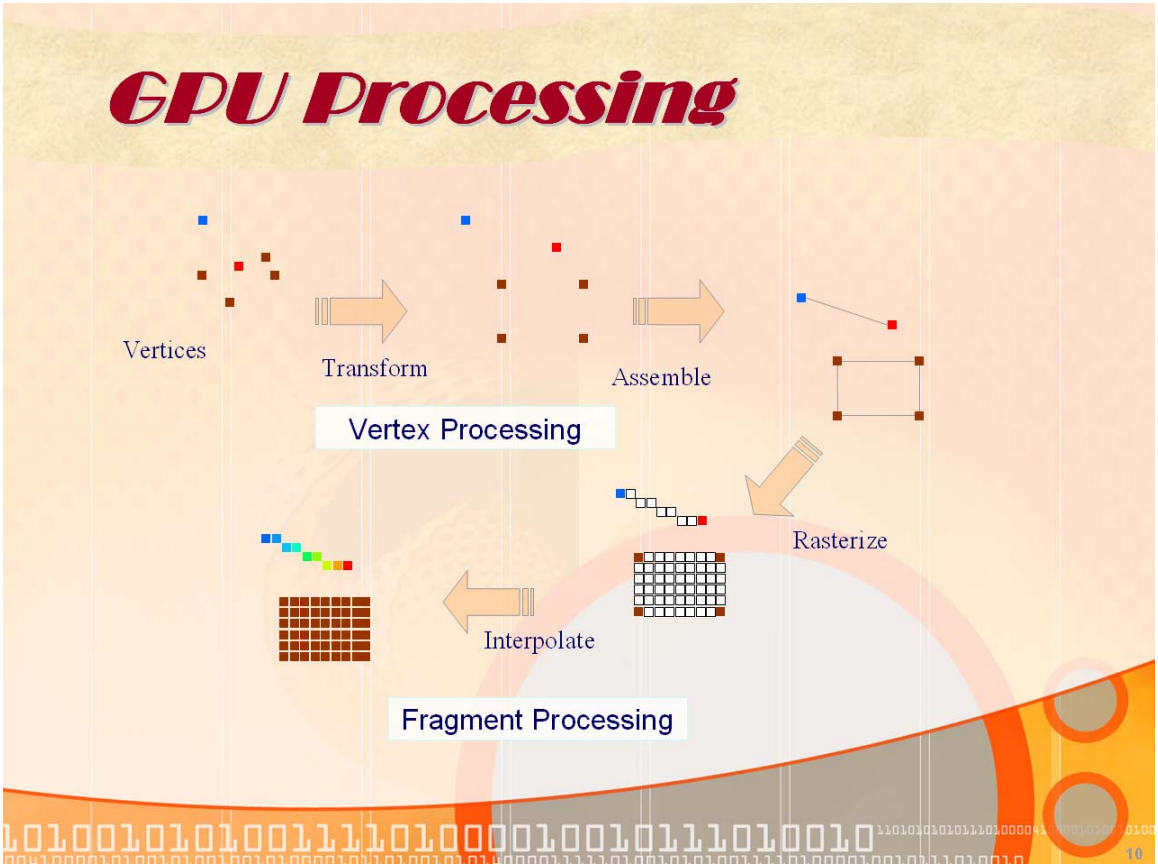

Figure 4 Typical steps in graphics processing on a GPU

First, vertices are input into a GPU and transformed according to the view port, and assembled. This processing is called vertex processing. Then, the GPU performs rasterization and interpolation to finish the image, called fragment processing. In this project, although we call ours 3D images, we only process 2D image data. In other words, we only focus on fragment processing program. The code is very easy to write and is shown in the Appendix. In this method, we still

have the problem of storing the filtered image data in graphic cards memory. With our cards we can store at most 9 images of less than 1024x1024 pixels. The memory bottleneck skews out performance results but is expected to be less of an issue with newer graphics cards that today employ even 512MB or onboard RAM.

We wrote our own convolution code using the shader language and tested it on the Nvidia card. We could not test this approach on the Wildcat card since it is an older card that does not support newer OpenGL functions exploited in the shader language implementation.

5. Results and Conclusions

The performance results for the CPU-based multiview rendering as well as for the GPU-based rendering under OpenGL and under shader language is shown in Table 1 below.

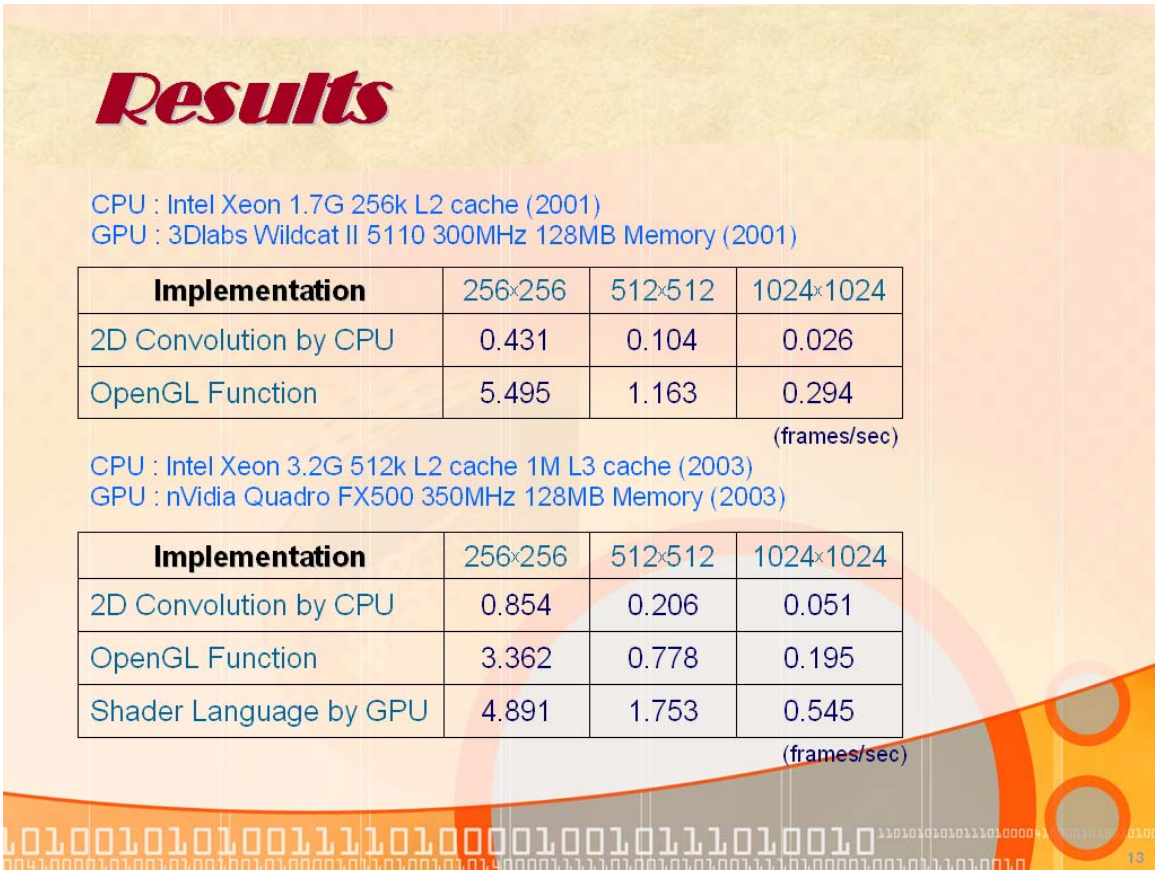

Table 1 Experimentally-measured performance in frames/sec for various implementations of multiview rendering.

Based on the performance results in Table 1 we can conclude that:

a. CPU-based multiview rendering is very slow and is very far from real-time performance.

b. GPU-based multiview rendering using OpenGL is up to an order of magnitude faster than the CPU-based rendering. Although the 3Dlabs Wildcat II 5110 is a fairly old card, it is a high-performance card that easily outperforms the newer but entry-level Nvidia card. Still, the performance of the Wildcat card is not real-time, especially for larger images. The

performance would be even slower at 1600x1200 pixels, resolution of our “Synthagram” SG202 screen for which the rendering was intended.

c. The GPU-based rendering exploiting our own convolution implemented using the shader language works well and is faster than the OpenGL implementation by a factor of about 2-3. It seems that this performance gain increases with the increasing images size.

d. Combining the OpenGL and shader language solutions, we believe it should be possible to reach real-time performance for 256x256-pixel images on newer high-end cards. In the comparison table, we only focused on improving the convolution speed in order to make the comparison fair (the CPU-based method cannot store images on the graphics card). Also, not that the sub-sampling pattern did not consist of binary numbers (0 or 1), and this added complexity in multiplications. Should the memory size permit storage of filtering results on the graphics card and should binary sub-sampling masks be permitted, the performance of the GPU approach could be further improved.

Appendix – C code

```
/*****************************************************************************************************************************
*
*                    SC913 REAL-TIME RENDERING OF MULTIPLE VIEWS USING OPENGL
*                                Chun-Wei Chan
*
*   There are three ways to do this algorithm implementation.
*   1. Convolution by CPU
*   2. Convolution by OpenGL function
*   3. Convolution by Shader Language
*
*   Usage : glconv.exe <method #> <input> <pattern> <Image Width> <Image Height> <# of Images>
*
*   <method #>     as listed above
*   <input>        Input file name, I combine 9 images into one RGB file. It is just for easy to bring
*              Users can easily use ImageMagic to build this. For example:
*                                                  convert Grille1.bmp -resize 256x256! pa1.bmp
*                                                  convert Grille2.bmp -resize 256x256! pa2.bmp
*                                                  convert Grille3.bmp -resize 256x256! pa3.bmp
*                                                  convert Grille4.bmp -resize 256x256! pa4.bmp
*                                                  convert Grille5.bmp -resize 256x256! pa5.bmp
*                                                  convert Grille6.bmp -resize 256x256! pa6.bmp
*                                                  convert Grille7.bmp -resize 256x256! pa7.bmp
*                                                  convert Grille8.bmp -resize 256x256! pa8.bmp
*                                                  convert Grille9.bmp -resize 256x256! pa9.bmp
*                                                  convert -append pa1.bmp pa2.bmp pa3.bmp pa4.bmp pa5.bmp
pa6.bmp pa7.bmp pa8.bmp pa9.bmp gr256.bmp
*                                                  convert gr256.bmp gr256b.rgb
*            <pattern>      Pattern file name
*            <Image Width>   Each single image's width
*            <Image Height>  Each single image's height
*            <# of images>   Number of images to be combined
*
*            Example:
*            glconv.exe 1 gr256b.rgb pat256b.rgb 256 256 9
*
*
*   And this program doesn't use any library for any specific OS, so this should be run well on Linux or MacOS X,
*   as long as their graphics card support OpenGL.
*
*****************************************************************************************************************************/

#include <iostream>
#include "windows.h"
#include <string.h>
#include "glew.h"    //for gl extension
#include "glut.h"     //insure using the lastest version, put glut in the same directory
#include <gl\glu.h>
#include <gl\gl.h>

// Due to some opengl functions not easy to put parameters in, so I set some global variables
float start,end;
int testFrames = 10;
float fps = 0;
```

```
int filterD = 7;          //filter dimension
float lpfilter[7][7]={ 0, 0, 0, 0, 0, 0, 0,
                                                0, 0, 0, 0, 0, 0, 0,
                                                0, 0, 0, 0, 0, 0, 0,
                                                0, 0, 0, 1, 0, 0, 0,
                                                0, 0, 0, 0, 0, 0, 0,
                                                0, 0, 0, 0, 0, 0, 0,
                                                0, 0, 0, 0, 0, 0, 0};
//float slpfilterr[7]={ 0, 0, 0, 1, 0, 0, 0};    //for separate filter
//float slpfilterc[7]={ 0, 0, 0, 1, 0, 0, 0};

float *lpfilterRGB, *slpfilterrRGB, *slpfiltercRGB;
int pindex = 0;
unsigned char* blank;
GLhandleARB   glProgObj;
GLhandleARB   fragShader;

typedef struct {
        char *file_name_in;
        char *file_name_pattern;
        char *file_name_out;
        int num_of_img;
        int sImgWidth;
        int sImgHeight;
        unsigned char *buf, *imgPtr, *imgOut, *imgTem, *patPtr;
        int slen;   //single frame size
        int len;
} imgSeqInfo;

typedef struct {
        int impMethod;       //implementation method
        int scrWidth;        //screen width
        int scrHeight;
        GLuint *tex;         //texture
} glInfo;

imgSeqInfo isi;
glInfo gli;

static const char *glsl2DConvolution = {
                        "uniform sampler2D tex, pat;"
                        "void main(void)"
                        "{"

                        //should manually change this for different size
                        "   const float offset = 1.0 / 256.0;"
                        //"   const float offset = 1.0 / 512.0;"
                        //"   const float offset = 1.0 / 1024.0;"

                        "           const float lpfilter[49] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,          0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,};"

                        /*
                        "                                                                                       0.0,  0.0,  0.0,  0.0,
0.0, 0.0, 0.0,"
```

```
                "                                                                                0.0,  0.0,  0.0,  0.0,
0.0, 0.0, 0.0,"
                "                                                                                0.0,  0.0,  0.0,  1.0,
0.0, 0.0, 0.0,"
                "                                                                                0.0,  0.0,  0.0,  0.0,
0.0, 0.0, 0.0,"
                "                                                                                0.0,  0.0,  0.0,  0.0,
0.0, 0.0, 0.0,"
                "                                                                                0.0,  0.0,  0.0,  0.0,
0.0, 0.0, 0.0,};"
                */
                "           vec2 texCoord = gl_TexCoord[0].xy;"
                "   vec4 c  = texture2D(tex, texCoord);"
                "   vec4 f1 = texture2D(tex, texCoord + vec2(-offset*3, offset*3));"
                "   vec4 f2 = texture2D(tex, texCoord + vec2(-offset*2, offset*3));"
                "   vec4 f3 = texture2D(tex, texCoord + vec2(-offset*1, offset*3));"
                "   vec4 f4 = texture2D(tex, texCoord + vec2(                  0, offset*3));"
                "   vec4 f5 = texture2D(tex, texCoord + vec2( offset*1, offset*3));"
                "   vec4 f6 = texture2D(tex, texCoord + vec2( offset*2, offset*3));"
                "   vec4 f7 = texture2D(tex, texCoord + vec2( offset*3, offset*3));"

                "   vec4 f8 = texture2D(tex, texCoord + vec2(-offset*3, offset*2));"
                "   vec4 f9 = texture2D(tex, texCoord + vec2(-offset*2, offset*2));"
                "   vec4 f10 = texture2D(tex, texCoord + vec2(-offset*1, offset*2));"
                "   vec4 f11 = texture2D(tex, texCoord + vec2(                  0, offset*2));"
                "   vec4 f12 = texture2D(tex, texCoord + vec2( offset*1, offset*2));"
                "   vec4 f13 = texture2D(tex, texCoord + vec2( offset*2, offset*2));"
                "   vec4 f14 = texture2D(tex, texCoord + vec2( offset*3, offset*2));"

                "   vec4 f15 = texture2D(tex, texCoord + vec2(-offset*3, offset*1));"
                "   vec4 f16 = texture2D(tex, texCoord + vec2(-offset*2, offset*1));"
                "   vec4 f17 = texture2D(tex, texCoord + vec2(-offset*1, offset*1));"
                "   vec4 f18 = texture2D(tex, texCoord + vec2(                  0, offset*1));"
                "   vec4 f19 = texture2D(tex, texCoord + vec2( offset*1, offset*1));"
                "   vec4 f20 = texture2D(tex, texCoord + vec2( offset*2, offset*1));"
                "   vec4 f21 = texture2D(tex, texCoord + vec2( offset*3, offset*1));"

                "   vec4 f22 = texture2D(tex, texCoord + vec2(-offset*3, 0));"
                "   vec4 f23 = texture2D(tex, texCoord + vec2(-offset*2, 0));"
                "   vec4 f24 = texture2D(tex, texCoord + vec2(-offset*1, 0));"
                "   vec4 f25 = texture2D(tex, texCoord + vec2(                  0, 0));"
                "   vec4 f26 = texture2D(tex, texCoord + vec2( offset*1, 0));"
                "   vec4 f27 = texture2D(tex, texCoord + vec2( offset*2, 0));"
                "   vec4 f28 = texture2D(tex, texCoord + vec2( offset*3, 0));"

                "   vec4 f29 = texture2D(tex, texCoord + vec2(-offset*3, -offset*1));"
                "   vec4 f30 = texture2D(tex, texCoord + vec2(-offset*2, -offset*1));"
                "   vec4 f31 = texture2D(tex, texCoord + vec2(-offset*1, -offset*1));"
                "   vec4 f32 = texture2D(tex, texCoord + vec2(                  0, -offset*1));"
                "   vec4 f33 = texture2D(tex, texCoord + vec2( offset*1, -offset*1));"
                "   vec4 f34 = texture2D(tex, texCoord + vec2( offset*2, -offset*1));"
                "   vec4 f35 = texture2D(tex, texCoord + vec2( offset*3, -offset*1));"

                "   vec4 f36 = texture2D(tex, texCoord + vec2(-offset*3, -offset*2));"
                "   vec4 f37 = texture2D(tex, texCoord + vec2(-offset*2, -offset*2));"
                "   vec4 f38 = texture2D(tex, texCoord + vec2(-offset*1, -offset*2));"
```

```
                        "   vec4 f39 = texture2D(tex, texCoord + vec2(                    0, -offset*2));"
                        "   vec4 f40 = texture2D(tex, texCoord + vec2( offset*1, -offset*2));"
                        "   vec4 f41 = texture2D(tex, texCoord + vec2( offset*2, -offset*2));"
                        "   vec4 f42 = texture2D(tex, texCoord + vec2( offset*3, -offset*2));"

                        "   vec4 f43 = texture2D(tex, texCoord + vec2(-offset*3, -offset*3));"
                        "   vec4 f44 = texture2D(tex, texCoord + vec2(-offset*2, -offset*3));"
                        "   vec4 f45 = texture2D(tex, texCoord + vec2(-offset*1, -offset*3));"
                        "   vec4 f46 = texture2D(tex, texCoord + vec2(                    0, -offset*3));"
                        "   vec4 f47 = texture2D(tex, texCoord + vec2( offset*1, -offset*3));"
                        "   vec4 f48 = texture2D(tex, texCoord + vec2( offset*2, -offset*3));"
                        "   vec4 f49 = texture2D(tex, texCoord + vec2( offset*3, -offset*3));"

/* example for 3x3 filter
                        "   vec4 bl = texture2D(tex, texCoord + vec2(-offset, -offset));"
                        "   vec4 l  = texture2D(tex, texCoord + vec2(-offset,     0.0));"
                        "   vec4 tl = texture2D(tex, texCoord + vec2(-offset,  offset));"
                        "   vec4 t  = texture2D(tex, texCoord + vec2(    0.0,  offset));"
                        "   vec4 tr = texture2D(tex, texCoord + vec2( offset,  offset));"
                        "   vec4 r  = texture2D(tex, texCoord + vec2( offset,     0.0));"
                        "   vec4 br = texture2D(tex, texCoord + vec2( offset, -offset));"
                        "   vec4 b  = texture2D(tex, texCoord + vec2(    0.0, -offset));"
                        "    gl_FragColor =  lpfilter[4]*c + lpfilter[6]*bl + lpfilter[3]*l + lpfilter[0]*tl + lpfilter[1]*t + lpfilter[2]*tr +
lpfilter[5]*r + lpfilter[8]*br + lpfilter[7]*b;"
*/
                        "gl_FragColor =  f1*lpfilter[0]+ f2*lpfilter[1] + f3*lpfilter[2] + f4*lpfilter[3] + f5*lpfilter[4] + f6*lpfilter[5] +
f7*lpfilter[6] + f8*lpfilter[7] + f9*lpfilter[8] + f10*lpfilter[9] + f11*lpfilter[10] + f12*lpfilter[11] + f13*lpfilter[12] +
f14*lpfilter[13] + f15*lpfilter[14] + f16*lpfilter[15] + f17*lpfilter[16] + f18*lpfilter[17] + f19*lpfilter[18] + f20*lpfilter[19] +
f21*lpfilter[20] + f22*lpfilter[21] + f23*lpfilter[22] + f24*lpfilter[23] + f25*lpfilter[24] + f26*lpfilter[25] + f27*lpfilter[26] +
f28*lpfilter[27] + f29*lpfilter[28] + f30*lpfilter[29] + f31*lpfilter[30] + f32*lpfilter[31] + f33*lpfilter[32] + f34*lpfilter[33] +
f35*lpfilter[34] + f36*lpfilter[35] + f37*lpfilter[36] + f38*lpfilter[37] + f39*lpfilter[38] + f40*lpfilter[39] + f41*lpfilter[40] +
f42*lpfilter[41] + f43*lpfilter[42] + f44*lpfilter[43] + f45*lpfilter[44] + f46*lpfilter[45] + f47*lpfilter[46] + f48*lpfilter[47] +
f49*lpfilter[48];"
/*
"   gl_FragColor =  f1*lpfilter[0] + f2*lpfilter[1] + f3*lpfilter[2] + f4*lpfilter[3] + f5*lpfilter[4] + f6*lpfilter[5] + f7*lpfilter[6]"
"           f8*lpfilter[7] + f9*lpfilter[8] + f10*lpfilter[9] + f11*lpfilter[10] + f12*lpfilter[11] + f13*lpfilter[12] + f14*lpfilter[13] +
"
"           f15*lpfilter[14] + f16*lpfilter[15] + f17*lpfilter[16] + f18*lpfilter[17] + f19*lpfilter[18] + f20*lpfilter[19] +
f21*lpfilter[20] + "
"           f22*lpfilter[21] + f23*lpfilter[22] + f24*lpfilter[23] + f25*lpfilter[24] + f26*lpfilter[25] + f27*lpfilter[26] +
f28*lpfilter[27] + "
"           f29*lpfilter[28] + f30*lpfilter[29] + f31*lpfilter[30] + f32*lpfilter[31] + f33*lpfilter[32] + f34*lpfilter[33] +
f35*lpfilter[34] + "
"           f36*lpfilter[35] + f37*lpfilter[36] + f38*lpfilter[37] + f39*lpfilter[38] + f40*lpfilter[39] + f41*lpfilter[40] +
f42*lpfilter[41] + "
"           f43*lpfilter[42] + f44*lpfilter[43] + f45*lpfilter[44] + f46*lpfilter[45] + f47*lpfilter[46] + f48*lpfilter[47] +
f49*lpfilter[48];"
*/
                        "}"
"   vec4 p  = texture2D(pat, gl_TexCoord[1].xy);"
//"           gl_FragColor =  gl_FragColor * p / 255.0;"// doesn't work!
};

void loadImgFile(){
            FILE *fr, *frp;
```

```
	fr=fopen(isi.file_name_in,"rb");
	if(fr!=NULL){printf("\n Input File : %s",isi.file_name_in);
	}else{printf("Open <%s> Failed!\n",isi.file_name_in);}
	isi.buf = (unsigned char *)malloc(isi.len);
	fread(isi.buf, isi.len, 1, fr);

	frp=fopen(isi.file_name_pattern,"rb");
	if(frp!=NULL){printf("\n Pattern File : %s",isi.file_name_pattern);
	}else{printf("Open <%s> Failed!\n",isi.file_name_pattern);}
	isi.patPtr = (unsigned char *)malloc(isi.len);
	fread(isi.patPtr, isi.len, 1, frp);

	fcloseall();
}

//OpenGL image format is from bottom to top, so I do flip
void flipVertical(){
	isi.imgPtr = (unsigned char *)malloc(isi.len);
	int one_line_size = isi.sImgWidth * 3;
	for(int i = 0; i < isi.num_of_img; i++){
		isi.buf += isi.slen - one_line_size;
		for(int h = 0; h < isi.sImgHeight; h++){
			memcpy(isi.imgPtr, isi.buf, one_line_size);
			isi.imgPtr += one_line_size;
			isi.buf -= one_line_size;
		}
		isi.buf += isi.slen;
	}
	isi.imgPtr -= isi.len;
}

//for some cards, texture only can be stored 9 or less
//so I do not use that
void bindTextures(){
	gli.tex = (GLuint *)malloc(isi.num_of_img);
	glGenTextures(9, &gli.tex[0]);
	/*
	for (int loop=0; loop<2; loop++){
		//glActiveTexture(GL_TEXTURE0 + loop);
		glBindTexture(GL_TEXTURE_2D, gli.tex[loop]);
		glTexImage2D(GL_TEXTURE_2D, 0, 3, isi.sImgWidth, isi.sImgHeight, 0, GL_RGB,
GL_UNSIGNED_BYTE, isi.imgPtr);
		glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
		glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
		isi.imgPtr += isi.slen;
	}
	isi.imgPtr -= isi.len;
	*/
}

//method 1 Convolution by CPU
void origConv(){
	int h, v, fh, fv;
	int midf = (filterD+1)/2 - 1;
	int fvalue, ishift, fshift;
	float temR, temG, temB, tem;
```

```
		for( h=0;h<isi.slen;h++){
			isi.imgOut[h] = 0;
		}
		for( int ii=0;ii<isi.num_of_img;ii++){
			for(h = filterD; h < isi.sImgHeight-filterD; h++){
				for(v = filterD; v < isi.sImgWidth-filterD; v++){
					ishift = (h*isi.sImgWidth + v)*3;
					temR = 0;temG = 0;temB = 0;
					for(fh = 0;fh < filterD; fh++){
						for(fv = 0;fv < filterD; fv++){
							fvalue = lpfilter[fh][fv];
							fshift = ((h+fh-midf)*isi.sImgWidth + (v+fv-midf))*3;
							temR += fvalue*isi.imgPtr[fshift];
							temG += fvalue*isi.imgPtr[fshift + 1];
							temB += fvalue*isi.imgPtr[fshift + 2];
						}
					}
					// apply pattern
					tem = temR * ((float)isi.patPtr[ishift] / 255);
					if(tem>255) tem = 255;
					else if(tem<0) tem = 0;
					isi.imgTem[ishift] = (unsigned char)tem;

					tem = temG * ((float)isi.patPtr[ishift+1] / 255);
					if(tem>255) tem = 255;
					else if(tem<0) tem = 0;
					isi.imgTem[ishift+1] = (unsigned char)tem;

					tem = temB * ((float)isi.patPtr[ishift+2] / 255);
					if(tem>255) tem = 255;
					else if(tem<0) tem = 0;
					isi.imgTem[ishift+2] = (unsigned char)tem;

				}
			}
			isi.imgPtr += isi.slen;
			isi.patPtr += isi.slen;
			for( h=0;h<isi.slen;h++)
				isi.imgOut[h] += isi.imgTem[h];
			memcpy(isi.imgTem, blank, isi.slen);
		}
		isi.imgPtr -= isi.len;
		isi.patPtr -= isi.len;
}
/*

Method 2 Convolution by OpenGL function
for some old cards only extension can be used, such as 3Dlabs WildcatII
for newer cards, Convolution function is in ARB_Image
Build Filter -> Enable Filter ->   DrawPixels   ->  Disable Filter ->  ReadPixels -> store in Tmp
                                                                                    (do not swapbuffer)
            ||<----------------------------------loop---------------------------->||

*/
void call2DConvolutionFunc(){
		float tem = 0;
```

```
        int i;
        memcpy(isi.imgOut, blank, isi.slen);

        //for cards using extension
        glEnable(GL_CONVOLUTION_2D_EXT);
        //glEnable(GL_SEPARABLE_2D_EXT);

        //for cards using ARB_image
        //glEnable(GL_CONVOLUTION_2D);
        //glEnable(GL_SEPARABLE_2D);

        for(int n=0; n<isi.num_of_img; n++){
                glClear (GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT ) ;
                glRasterPos2f( -1.0, -1.0 );
                glDrawPixels( isi.sImgWidth, isi.sImgHeight, GL_RGB, GL_UNSIGNED_BYTE, isi.imgPtr);

                //for cards using extension
                glDisable(GL_CONVOLUTION_2D_EXT);
                //glDisable(GL_SEPARABLE_2D_EXT);

                //for cards using ARB_image
                //glDisable(GL_CONVOLUTION_2D);
                //glDisable(GL_SEPARABLE_2D);

                glReadPixels( 0, 0, isi.sImgWidth, isi.sImgHeight, GL_RGB, GL_UNSIGNED_BYTE, isi.imgTem);
                for(i=0; i<isi.slen; i++){
                        tem = (float)isi.imgTem[i] * ((float)isi.patPtr[i] / 255);
                        if(tem>255) tem=255;
                        else if(tem<0) tem = 0;
                        isi.imgTem[i] = (unsigned char)tem;
                        isi.imgOut[i] += isi.imgTem[i];
                }
                memcpy(isi.imgTem, blank, isi.slen);
                isi.imgPtr += isi.slen;
                isi.patPtr += isi.slen;
        }
        isi.imgPtr -= isi.len;
        isi.patPtr -= isi.len;

        //for cards using extension
        glDisable(GL_CONVOLUTION_2D_EXT);
        //glDisable(GL_SEPARABLE_2D_EXT);

        //for cards using ARB_image
        //glDisable(GL_CONVOLUTION_2D);
        //glDisable(GL_SEPARABLE_2D);

        glClear (GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT ) ;
        glRasterPos2f( -1.0, -1.0 );
        //glRasterPos2f( -1.0 + ((float)pindex*20/isi.sImgWidth), -1.0 );          test if it works well
        glDrawPixels( isi.sImgWidth, isi.sImgHeight, GL_RGB, GL_UNSIGNED_BYTE, isi.imgOut);
}

void display (){
        glClear (GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT ) ;
        if(gli.impMethod == 1){
```

```
                origConv();
                glRasterPos2f( -1.0, -1.0 );
                //glRasterPos2f( -1.0 + ((float)pindex*20/isi.sImgWidth), -1.0 );          test if it works well
                glDrawPixels( isi.sImgWidth, isi.sImgHeight, GL_RGB, GL_UNSIGNED_BYTE, isi.imgOut);
        }else if(gli.impMethod == 2){
                call2DConvolutionFunc();
        }else if(gli.impMethod == 3){
                glEnable(GL_TEXTURE_2D);
                //Build Shader Language Link
                glProgObj = glCreateProgramObjectARB();
                fragShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
                glShaderSourceARB(fragShader, 1, &glsl2DConvolution, NULL);
                glCompileShaderARB(fragShader);
                glAttachObjectARB(glProgObj, fragShader);
                glLinkProgramARB(glProgObj);
                GLint progLinkSuccess;
                glGetObjectParameterivARB(glProgObj,                    GL_OBJECT_LINK_STATUS_ARB,
        &progLinkSuccess);
                if (!progLinkSuccess){
                        printf("Filter shader could not be linked\n");
                        exit(1);
                }
                //End Build Link
                for(int convLoop = 0; convLoop<isi.num_of_img; convLoop++){
                        glActiveTexture(GL_TEXTURE0);
                        glBindTexture(GL_TEXTURE_2D, gli.tex[0]);
                        glTexImage2D(GL_TEXTURE_2D,  0,  3,  isi.sImgWidth,  isi.sImgHeight,  0,  GL_RGB,
GL_UNSIGNED_BYTE, isi.imgPtr);
                        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
                        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                        isi.imgPtr += isi.slen;

                        glUniform1iARB(glGetUniformLocationARB(glProgObj, "tex"), convLoop);

                        glUseProgramObjectARB(glProgObj);

                        glBegin(GL_QUADS);
                        glTexCoord2f(0, 0); glVertex2f(-1, -1);
                        glTexCoord2f(1, 0); glVertex2f( 1, -1);
                        glTexCoord2f(1, 1); glVertex2f( 1,  1);
                        glTexCoord2f(0, 1); glVertex2f(-1,  1);
                        glEnd();

                        glUseProgramObjectARB(0);
                        //glActiveTextureARB(GL_TEXTURE0_ARB + convLoop);
                        glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, isi.sImgWidth, isi.sImgHeight);

                        glBegin(GL_QUADS);
                        glTexCoord2f(0, 0); glVertex2f(-1, -1);
                        glTexCoord2f(1, 0); glVertex2f( 1, -1);
                        glTexCoord2f(1, 1); glVertex2f( 1,  1);
                        glTexCoord2f(0, 1); glVertex2f(-1,  1);
                        glEnd();
                        glReadPixels(0,  0,  isi.sImgWidth,  isi.sImgHeight,  GL_RGB,  GL_UNSIGNED_BYTE,
isi.imgTem);
                        float tem = 0;
```

```
                        for(int i=0; i<isi.slen; i++){
                                tem = (float)isi.imgTem[i] * ((float)isi.patPtr[i] / 255);
                                if(tem>255) tem=255;
                                if(tem<0) tem = 0;
                                isi.imgTem[i] = (unsigned char)tem;
                                isi.imgOut[i] += isi.imgTem[i];
                        }
                        memcpy(isi.imgTem, blank, isi.slen);
                        isi.patPtr += isi.slen;
                }
                glDisable(GL_TEXTURE_2D);
                glClear (GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT ) ;
                glRasterPos2f( -1.0, -1.0 );
                glDrawPixels( isi.sImgWidth, isi.sImgHeight, GL_RGB, GL_UNSIGNED_BYTE, isi.imgOut);
                isi.imgPtr -= isi.len;
                isi.patPtr -= isi.len;
        }
        glutSwapBuffers();
}

void keyboard(unsigned char key, int x, int y){
        switch(key){
        case 27:
                exit(0);
                break;
        default:
                pindex = 0;
                start = glutGet(GLUT_ELAPSED_TIME) * 0.001;
                printf("\nStart Time = %f", start);
                for(int i = 0; i<testFrames; i++){
                        display();
                        end = glutGet(GLUT_ELAPSED_TIME) * 0.001;
                        printf("\nDisplay Frame # %d  Time = %f", i+1, end);
                        pindex++;
                }
                fps=testFrames/(end-start);
                printf("\nFPS = %f   (%d frames)\n", fps, testFrames);
        }
}
void set2DConvolutionFilter(){
        lpfilterRGB = (float *)malloc(filterD*filterD*3);
        //for separate filter
        //slpfilterrRGB = (float *)malloc(filterD*3);
        //slpfiltercRGB = (float *)malloc(filterD*3);
        for(int i=0; i<filterD; i++){
                for(int j=0; j<filterD; j++){
                        lpfilterRGB[(i*filterD+j)*3] = lpfilter[i][j];
                        lpfilterRGB[(i*filterD+j)*3+1] = lpfilter[i][j];
                        lpfilterRGB[(i*filterD+j)*3+2] = lpfilter[i][j];
                }
/*      for separate filter
                slpfilterrRGB[(i*filterD+i)*3] = slpfilterr[i];
                slpfilterrRGB[(i*filterD+i)*3+1] = slpfilterr[i];
                slpfilterrRGB[(i*filterD+i)*3+2] = slpfilterr[i];
                slpfiltercRGB[(i*filterD+i)*3] = slpfilterc[i];
                slpfiltercRGB[(i*filterD+i)*3+1] = slpfilterc[i];
```

```
                slpfiltercRGB[(i*filterD+i)*3+2] = slpfilterc[i];
*/
        }

        // for cards using extension
        glConvolutionFilter2DEXT( GL_CONVOLUTION_2D, GL_RGB, filterD, filterD, GL_RGB, GL_FLOAT,
lpfilterRGB);
        //glSeparableFilter2DEXT( GL_SEPARABLE_2D_EXT, GL_RGB, filterD, filterD, GL_RGB, GL_FLOAT,
slpfilterrRGB, slpfiltercRGB);
        //glEnable(GL_CONVOLUTION_2D_EXT);

        // for cards using ARB_image
        //glConvolutionFilter2D( GL_CONVOLUTION_2D, GL_RGB, filterD, filterD, GL_RGB, GL_FLOAT,
lpfilterRGB);
        //glSeparableFilter2D( GL_SEPARABLE_2D, GL_RGB, filterD, filterD, GL_RGB, GL_FLOAT, slpfilterrRGB,
slpfiltercRGB);
        //glEnable(GL_CONVOLUTION_2D);
}

void setDefault() {
        gli.impMethod = 2;
        isi.num_of_img = 9;
        /*
        if(gli.impMethod < 3){
                isi.file_name_in = "gr640.rgb";
                isi.file_name_out = "gr640out.rgb";
                isi.file_name_pattern = "pat640.rgb";
                isi.sImgWidth = 640;
                isi.sImgHeight = isi.sImgWidth * 0.75;
                //isi.sImgHeight = 144;
        }else{*/
                isi.file_name_in = "gr256b.rgb";
                isi.file_name_out = "gr256bout.rgb";
                isi.file_name_pattern = "pat256b.rgb";
                isi.sImgWidth = 256;
                isi.sImgHeight = isi.sImgWidth;
        //}
}

void setInit(){
        gli.scrWidth = isi.sImgWidth;
        gli.scrHeight = isi.sImgHeight;
        isi.slen = isi.sImgWidth * isi.sImgHeight * 3;
        isi.len = isi.slen * isi.num_of_img;
        isi.imgOut = (unsigned char *)malloc(isi.slen);
        isi.imgTem = (unsigned char *)malloc(isi.slen);
        blank = (unsigned char *)malloc(isi.slen);
        for(int i=0;i<isi.slen;i++){
                blank[i]=0;
        }
        memcpy(isi.imgOut, blank, isi.slen);
        memcpy(isi.imgTem, blank, isi.slen);
}

void main (int argc, char** argv)          {
        setDefault();
```

```
	if (argc>=8)		isi.file_name_out = argv[7];
	if (argc>=7)		isi.num_of_img = atoi(argv[6]);
	if (argc>=6)		isi.sImgHeight = atoi(argv[5]);
	if (argc>=5)		isi.sImgWidth = atoi(argv[4]);
	if (argc>=4)		isi.file_name_pattern = argv[3];
	if (argc>=3)		isi.file_name_in = argv[2];
	if (argc>=2)		gli.impMethod = atoi(argv[1]);

	setInit();

	glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH) ;
	glutInitWindowSize (gli.scrWidth, gli.scrHeight) ;
	glutInitWindowPosition (350,1) ;
	glEnable( GL_DEPTH_TEST );

	loadImgFile();
	flipVertical();
	if(gli.impMethod == 1){
		gli.win[0] = glutCreateWindow(" ~ 2D Convolution using openGL ~  2D Convolution by CPU");
	}else if(gli.impMethod == 2){
		gli.win[0] = glutCreateWindow(" ~ 2D Convolution using openGL ~  2D Convolution by openGL
Function");
		GLenum err = glewInit();
		if(GLEW_OK != err)
			printf("Error: %s\n", glewGetErrorString(err));
		if(glewIsSupported("GL_EXT_convolution"))
			printf("GL_EXT_convolution is supported! \n");
		else
			printf("GL_EXT_convolution isn't supported!\n");
		set2DConvolutionFilter();
	}else if(gli.impMethod == 3){
		gli.win[0] = glutCreateWindow(" ~ 2D Convolution using openGL ~  2D Convolution by Shader
Language");
		GLenum err = glewInit();
		if(GLEW_OK != err)
			printf("Error: %s\n", glewGetErrorString(err));
		if(glewIsSupported("GL_EXT_convolution"))
			printf("GL_EXT_convolution is supported! \n");
		else
			printf("GL_EXT_convolution isn't supported!\n");
		bindTextures();
	}
	glutDisplayFunc (display) ;
	glutKeyboardFunc(keyboard);
	glutMainLoop () ;
}
```