

## CIS 665 – GPU Programming and Architecture

Homework #3

Due: June 6/09/09 : 23:59:59PM EST

### 1) Benchmarking your GPU (25 points)

#### **Background:**

**GPUBench** is a benchmark suite designed to analyze the performance of programmable graphics processors in areas of particular importance to general purpose computation. Included in the suite are tests that rigorously exercise a system's GPU, determining statistics such as memory input/output bandwidth to floating point buffers, texture cache bandwidth, data download and readback rates, instruction throughput, and instruction precision. More so than being a measuring stick for GPU performance, GPUBench is intended to be a tool for developers of GPU-accelerated applications, since a more complete understanding of machine capabilities will dictate better software design decisions.

#### Extra Credit (5 Points) :

Rather than run the subset of tests in the “Problem” section below which are required. You can run all the pearl scripts instead in a similar fashion to the library of results. And create the graphs for your report and copy the graphs from the library And note any differences.

(NOTE: This is not a requirement though, and there is a ton of extra credit, some people may or may not find running the pearl scripts easier. If it confuses you, don't spend more than 1 minute trying to figure this extra credit part out, just work on the problem section right below, its simple enough to just copy and paste the commands into the command line and make a graph in latex or work, **we rather you work on problem 2 and 3** and budget your time accordingly rather than figure out how to run a pearl script)

#### **Problem:**

First download GPUBench: <http://graphics.stanford.edu/projects/gpubench/>

Second, take a look at “the library of results”:  
<http://graphics.stanford.edu/projects/gpubench/results/>

Compose a document called: GPUBenchResults.pdf or GPUBenchResults.doc that includes:

Your Name:

CIS 665 : Homework 3

(and includes the following sections)

a) **Discussion:** Look at the results for the 8800GTX-0003 vs 7900GTX-9371 Briefly discuss in a couple paragraphs what you think the major improvements are between the generations of hardware citing technical information the 2 GPU bench results show to back up your case.

b) **Video Driver Information:** Run from the command line, in the bin directory the “gid” and copy the results. Just type gid and it will spill out information once your in the path.

b) **Streaming: Basic Throughput :** Run the following tests on your card, look at the results of the closest card in the library results and compare in a 2 column table (fancy graphs not needed) the results

A comparison of pixel throughput when executing a simple shader program that fetches once from a floatN texture, performs a few ADD operations, and outputs the result to a floatN buffer.

```
fpfilltest -p -n -c 1  
fpfilltest -p -n -c 2  
fpfilltest -p -n -c 3  
fpfilltest -p -n -c 4
```

```
fpfilltest -b -n -c 1  
fpfilltest -b -n -c 2  
fpfilltest -b -n -c 3  
fpfilltest -b -n -c 4
```

(**Note:** Both tests represent the same test. One shows MPix/sec and one GB/sec.)

A comparison of performance obtained when issuing a shader using either a screen covering triangle or a large quad. The shader reads from a (1 or 4-component) float texture, performs a few ADD instructions, and outputs the result to a (1 or 4-component) pBuffer.

```
fpfilltest.exe -n -c 1 -r triangle  
fpfilltest.exe -n -c 1 -r quad  
fpfilltest.exe -n -c 4 -r triangle  
fpfilltest.exe -n -c 4 -r quad
```

Measures readback performance of `glReadPixels(0, 0, 512, 512, FORMAT, TYPE, ptr)` from a single buffered window and 4-component float pbuffer.

```
readback.exe -x -r  
readback.exe -x -b  
readback.exe -f -r  
readback.exe -f -b
```

(If you numbers are very different here then the library of results guess why)

Measures rate at which texture data can be loaded onto the card using multiple calls to `glTexSubImage2D()`. The texture is a 512x512 n-component float texture.

```
download.exe -n -c 1  
download.exe -n -c 2  
download.exe -n -c 3  
download.exe -n -c 4
```

Measures rates at which various shader instructions can be executed. Vector instructions are executed with 4-component vector operands.

```
instrissue -n -a -l 64 -m
```

(Here the second column and the third column are the interesting results to compare)

Compares the rate a which the card can perform ADD, SUB, MUL, and MAD instructions when operating on both scalar and 4-component vector operands. Any dual issue capability in the hardware should be apparent when performing the single component version of the instructions.

```
instrissue -n -c 1 -i ADD -l 40 -m  
instrissue -n -c 4 -i ADD -l 40 -m  
instrissue -n -c 1 -i SUB -l 40 -m  
instrissue -n -c 4 -i SUB -l 40 -m  
instrissue -n -c 1 -i MUL -l 40 -m  
instrissue -n -c 4 -i MUL -l 40 -m  
instrissue -n -c 1 -i MAD -l 40 -m  
instrissue -n -c 4 -i MAD -l 40 -m
```

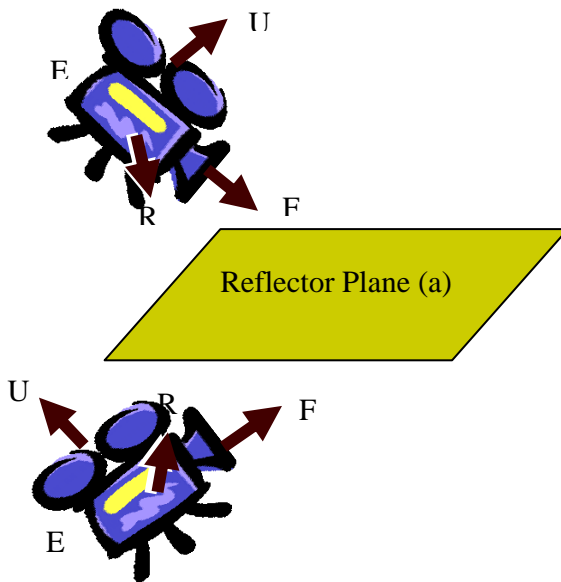
## 2) Reflections on Surfaces (50 points)

### Background:

Reflections allow added realism to be created for complex scenes. There are many techniques for producing reflections ranging from ray-tracing to cub-maps. Some of these techniques, such as ray-tracing, are more computationally expensive than others. We are going to implement a simple reflective surface. In order to accomplish this we will use environment mapping (EM). EM is a common technique where we assume the reflected environment is infinitely far away from the reflected object.

In order to accomplish this we reflect the current viewpoint about the reflector plane and then render the relevant parts of the scene from this point. We must make sure to orient the camera correctly in order to avoid culling issues. Assume the reflective surface is contained in the plane  $a$ . If the viewpoint/camera position is  $E$  and the forward, right and up vectors are  $F$ ,  $R$ , and  $U$ , then:

$$E_r = \text{reflect}_a(E), F_r = \text{reflect}_a(F), R_r = \text{reflect}(R) \text{ and } U_r = -\text{reflect}_a(U)$$



In order to perform the reflection we must calculate our reflection texture lookup for an incoming fragment with world coordinates  $F_{\text{pos}}$ .

1. Multiply  $F_{\text{pos}}$  by the reflected viewpoint,  $V_r$ .
2. Then multiply the result by the projection matrix  $P$  to get the post-projection space.
3. Scale the coordinates to the  $[-1,1]$  to  $[0,1]$  range ( $S$ ).
4. Use the result for the texture lookup.  $(S*P*V_r*F_{\text{pos}}).xy$
5. Note:  $S*P*V_r$  can be computed on the CPU and passed to your fragment program.

**Problem:**

Modify the hello world example on the course website to begin your implementation. The hello world example shows how to perform a multipass rendering using a frame buffer object (FBO) in CG. Your example will require two passes. In the first pass you will render the scene from the viewpoint of the reflected plane  $V_r$  as described above. In the second pass you will re-render the scene from the camera viewpoint and also render the reflected surface

First add a depth render buffer to the example scene as described in lecture 2.

```
// Create the render buffer
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, _depth_rb);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT24,
_iWidth, _iHeight);
// Attach the render buffer to the frame buffer
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
                             GL_DEPTH_ATTACHMENT_EXT,
                             GL_RENDERBUFFER_EXT, _depth_rb);
```

Note: the render buffer `_depth_rb` is a global variable of type `GLuint`. Remember the purpose of the render buffer is to ensure that depth is calculated when rendering to the frame buffer. If you would like to see the results of the frame buffer, later try viewing the rendered result in the frame buffer with the render buffer turned on or off.

Now we are ready to render the scene from the viewpoint of the reflector. First modify the viewpoint of the camera to give a more interesting view than from straight on. Next compute the reflected viewpoint as described above. In the update function modify the viewpoint to the reflected viewpoint, render the scene to the frame buffer, and return the viewpoint to its original configuration.

Next modify the fragment program to produce the correct texture lookup as described above. Note: this can either be done in the update function by rendering to a second texture, as originally set up, and then performing a standard texture mapping to the plane in the display function, or performing the fragment program on the plane directly in the display function.

In the display function render the plane that the reflection will be displayed upon, and re-render the original scene.

Note: In order to make the scene more interesting you may wish to change the colors of the objects in the scene, or implement standard OpenGL lighting to produce a more dynamic result. You can also easily texture map the plane by downloading an OpenGL image loader from the internet. In order to merge the texture with the frame buffer texture simply interpolate between the two:  $result = (textureColor + reflectedColor) / 2$ .

### 3) Matrix operations on the GPU using CG (20 points)

A) Extending your addition and subtraction GPGPU code. Implement a GPU program for multiplying two dense  $n \times m$  matrices. Use the scheme that employs  $n^2$  memory, runs in  $n/b$  passes,  $b$  represents the number of arithmetic operations you perform per pass. For simplicity, load your data in one channel

Find the value of  $b$  that maximizes the efficiency of the resulting program and report it for both operations. Including what card you are using.

The result should be written to a file result.txt, matrices should be read from a file and have the following form:

---

#### Extra Credit 1: Matrix Multiplication extension (10 points)

Implement a GPU program for multiplying two dense  $n \times m$  matrices that takes advantage of the GPU's inherent SIMD control by using all the channels (RGBA) and swizzling and smearing techniques discussed in class. Again find the value of  $b$  that maximizes the efficiency.

#### Extra Credit 2: Parallel Reductions (10 points)

Implement a GPU program for adding all the numbers in a 2D  $n \times n$  texture in  $O(\log n)$  passes.

#### Extra Credit 3: Bumpy Reflections (10 points)

Modify your result in problem three to create a bumpy reflection surface as described in the article Reflections from Bumpy Surfaces in the book Shader X3. The technique will be explained in lecture three and a copy of the article will be given out.

### References

1. C. Everitt. *Order-Independent Transparency*.  
[http://developer.nvidia.com/object/Interactive\\_Order\\_Transparency.html](http://developer.nvidia.com/object/Interactive_Order_Transparency.html)
2. Henning Mortveit, Reflections from Bumpy Surfaces. Shader X3. Charles River Media, Inc. 2005.