CSCI 4110 Laboratory Three

Performance

Introduction

This laboratory is devoted to the problem of performance. The first thing we will examine is reading OBJ files and how that can be improved. The second thing we will look at is using interleaved vertex array objects. This will lead to an examination of some of the tools that are available for monitoring performance.

OBJ Files

The OBJ file format is great for distributing files, it is an ASCII format that can easily be moved over the Internet. It is also fairly easy to read and write and quite a number of libraries have been developed for this. For small models reading and writing the files is fairly quick, but for larger models it can be very slow. We quite often like to have larger models to test our programs, and for commercial applications we need to be able to read large models quickly. This part of the laboratory uses the Buddha model that has over a million triangles. This model is over 90Mbytes of text, so it take some time to read. If we are debugging a program or running multiple tests, this quickly becomes annoying. The solution to this problem is to read the model once as an OBJ file and then save a binary representation that can be used in future runs of the program. Reading the binary data is much faster making our lives much easier.

There are two key questions that need to be answered at this point:

1. What format should we use for the binary file?
2. How do we know when the binary file exists so we don't need to read the OBJ file?

When we read the OBJ files we construct the vertices, normals and indices arrays, these are the natural candidates for writing to the binary file. We also need to write the size of these arrays at the start of the file so we know how many bytes to read.

For the second question there is a _stat function that can be called to determine if a file exists and some properties of the file. The first parameter to this function is the path to the file and the second parameter is a buffer where the file properties can be saved. This procedure returns 0 if the file has been found, which is the only information that we need.

We now have everything we need to start modifying the program from laboratory two. Start by copying all the files from laboratory two and download the budda.zip file from Blackboard (it is in the resources folder and it's large). You will need to unzip budda.zip and copy the budda.obj file to the same folder as the executable for the program. Now we are ready to start making changes to the program code.

To use the _stat procedure and the raw read and write code we need a number of include files, which are:

```
#include <fcntl.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <io.h>
```

All the changes are in the init procedure that we originally wrote for the sphere. Just after the creation of the vertex array object we add the following code:

```
result = _stat("buddha.bin", &buf);
if (result == 0) {
```

Where the declaration of buf is:

```
struct _stat buf;
```

We will start with the else part of this if statement. This includes the previous code that we had for reading the OBJ file and storing the result in the vertices, normals and indices arrays. Remember to change the input file from sphere.obj to budda.obj. After this code we add the following lines to our program:

```
fid = _open("buddha.bin", _O_WRONLY | _O_BINARY | _O_CREAT, _S_IREAD | _S_IWRITE);

result = _write(fid, &nv, (sizeof vertices));
result = _write(fid, &nn, (sizeof normals));
result = _write(fid, &ni, (sizeof indices));

result = _write(fid, vertices, nv*(sizeof GLfloat));
result = _write(fid, normals, nn*(sizeof GLfloat));
result = _write(fid, indices, ni*(sizeof GLuint));

_close(fid);
```

Where fid and result are integer variables. There are two things to note in the _open call. First, we must use the _O_BINARY flag otherwise Windows will assume we are writing a text file and will insert extra characters into the file for us. Second, we create the budda.bin file, since the _stat call told us it doesn't exist. The third parameter to _open are the permissions associated with the file. The three parameters to the _write call are the file identifier (from _open), the buffer of data to write and the size of the buffer. The _write procedure returns the number of bytes that were actually written, which should be the same as the third parameter.

Now that we know what's been written we can return to the then part of the if statement. The code for this is:

```
fid = _open("buddha.bin", _O_RDONLY | _O_BINARY);

result = _read(fid, &nv, (sizeof vertices));
result = _read(fid, &nn, (sizeof normals));
result = _read(fid, &ni, (sizeof indices));
triangles = ni / 3;
printf("v: %d, n: %d i: %d\n", nv, nn, ni);

vertices = new GLfloat[nv];
result = _read(fid, vertices, nv*(sizeof GLfloat));
normals = new GLfloat[nn];
result = _read(fid, normals, nn*(sizeof GLfloat));
indices = new GLuint[ni];
result = _read(fid, indices, ni*sizeof(GLuint));
```

```
_close(fid);
```

Again when we open the file we specify _O_BINARY so character translations aren't performed on our data. We start by reading the sizes of the three arrays and then printing the result. This is a simple check to see if the file has been written correctly. This is something that you would only use during debugging. For each of the three arrays we first allocate space for the array and then read in the contents of the array. The first parameter to _read is the file identifier, the second parameter is the location where the data will be saved and the third parameter is the number of bytes to read.

After you have made these changes run the program twice. The first run will read the budda.obj file and will take some time before the model is displayed. The second run will read budda.bin and will start displaying the model almost instantly. Take a look at the size of the budda.bin file, this is the amount of information that we are sending to the GPU.

Interleaved Vertex Array Buffers

The next change that we will make to the program is to use interleaved buffers. In our programs so far we have used separate arrays for the vertices and normal vectors. With interleaving we place the vectors and normal vectors in the same array with the normal vector for a vertex placed just after the vertex. This should result in better performance since it makes better use of DRAM.

We already have all the data in the vertices and normals array it is only a matter of repackaging it. As in the previous laboratory we will use the defined constant INTERLEAVED to control whether the vertex array buffer is interleaved, so add the following close to the start of your program:

```
#define INTERLEAVED
```

Commenting out this define will return the program to its original behavior. The code for the interleaving is:

```
#ifdef INTERLEAVED
combined = new GLfloat[nv + nn];
k = 0;
for(i=0; i<nv/3; i++) {
       combined[k++] = vertices[3*i];
       combined[k++] = vertices[3*i+1];
       combined[k++] = vertices[3*i+2];
       combined[k++] = normals[3*i];
       combined[k++] = normals[3*i+1];
       combined[k++] = normals[3*i+2];
}
glGenBuffers(1,&vbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vbuffer);
glBufferData(GL_ARRAY_BUFFER, (nv + nn)*sizeof(GLfloat), combined, GL_STATIC_DRAW);

glUseProgram(program);
vPosition = glGetAttribLocation(program, "vPosition");
glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 6*(sizeof GLfloat), 0);
glEnableVertexAttribArray(vPosition);
```

```
vNormal = glGetAttribLocation(program, "vNormal");
glVertexAttribPointer(vNormal, 3, GL_FLOAT, GL_FALSE, 6*(sizeof GLfloat),
(void*)(combined+3*(sizeof GLfloat)));
glEnableVertexAttribArray(vNormal);
```

The variable combined is a GLfloat pointer the same as vertices and normals. We start by allocating enough space for the vertices and normals. The for loop copies the (x,y,z) coordinates of a vertex following the (x,y,z) coordinates of its normal vector to the combined array. We then created a new GL_ARRAY_BUFFER and copy the combined data into it. In this case we can use the glBufferData to copy the data into the buffer since there is a single array containing all the data.

Next we need to set up the mapping from the buffer to the vertex attributes. This is basically the same as before except for a few changes in the glVertexAttribPointer procedure calls. First we need to set the stride parameter. In our previous programs this was set to 0, but now it needs to be set to the distance between vertex entries. There a 3 floats per vertex and 3 floats per normal, so the stride is 6 floats. This needs to be converted to bytes by multiplying 6 by (sizeof GLfloat). For the sixth parameter the vertices still start at the beginning of the array, so we use 0 for this parameter in the first call. For the normal vectors we need to skip the first three floats, so the sixth parameter in the second call is the start of the array plus the size of three floats.

Our previous code for setting up the vertex array buffers is place inside #else and #endif (on separate lines) just after the code we have just added. Note there is no change to the GL_ELEMENT_ARRAY_BUFFER, so this code should be outside of the #ifdef.

We want to compare interleaved versus non-interleaved vertex array buffers so we need to add some timing code to our program. First we need a function that provides an accurate time value. Thanks to Intel we have the following function:

```
double getSeconds() {
	LARGE_INTEGER freq, val;
	QueryPerformanceFrequency(&freq);
	QueryPerformanceCounter(&val);
	return (double)val.QuadPart / (double)freq.QuadPart;
}
```

This returns a floating point version of the most accurate clock on Windows. We need to time the glDrawElements call in our display procedure. This is done in the following way:

```
glBindVertexArray(objVAO);
t1 = getSeconds();
glDrawElements(GL_TRIANGLES, 3*triangles, GL_UNSIGNED_INT, NULL);
glFinish();
printf("%f\n", getSeconds() - t1);
```

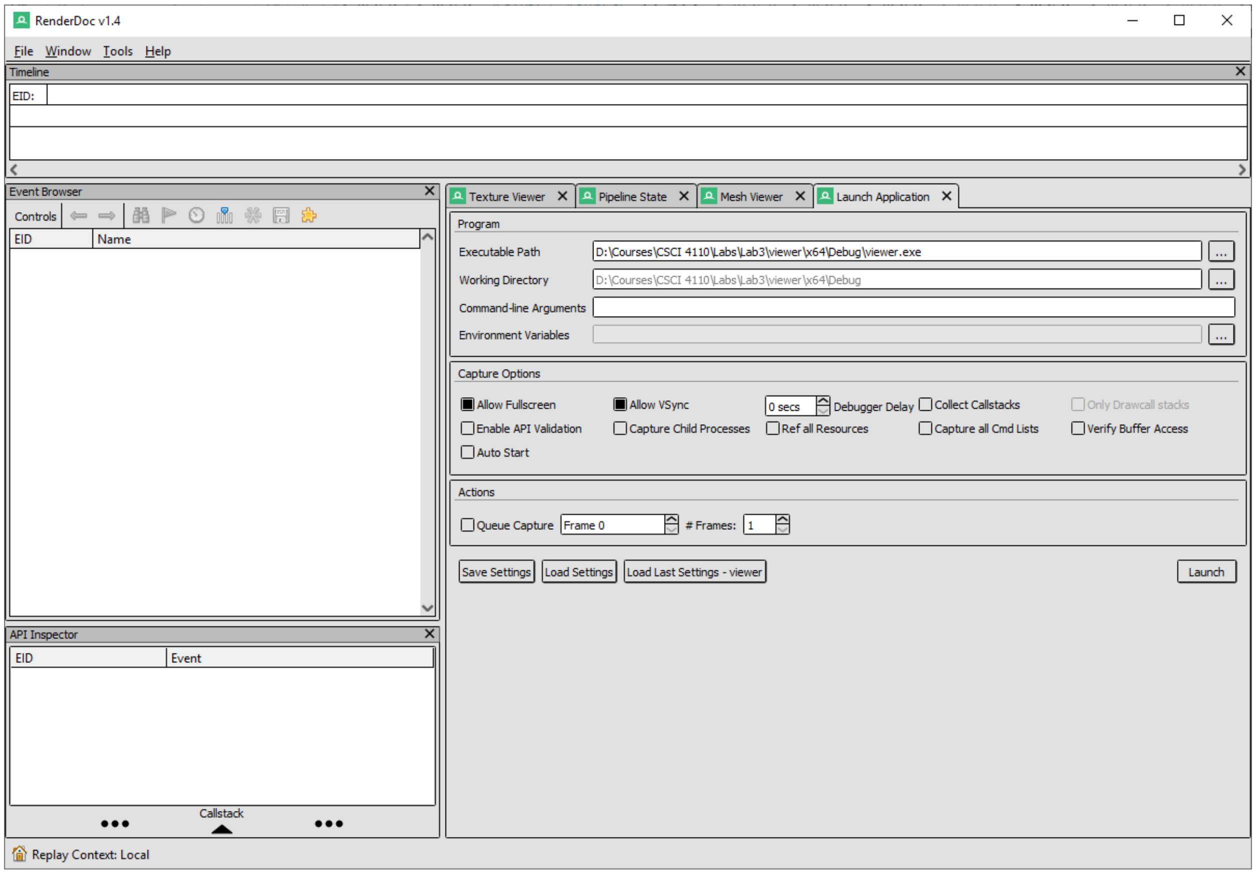
The glFinish() procedure waits until the GPU is finished before it returns. Normally glDrawElements returns immediately, so if we don't do this we won't get meaningful timing results.

Make the changes described above to the program and run it with INTERLEAVED defined and the define statement commented out. The timing results you get should be a bit surprising. Both runs of the program will produce basically the same times. It appears that there is no difference between using interleaved and non-interleaved vertex array buffers. This calls for a bit more investigation.

Performance Analysis Tools

To determine what's going on we need to use one of the performance analysis tools. Each of the GPU chip manufacturers provides a performance tool for their GPUs which can be used to determine how well their GPUs are performing. At least the basic levels of these tools are free and easy to download. The main problem is there isn't a tool that is OS and GPU independent. The ones from AMD and NVidia only work reliably with their GPUs, and the one from Intel doesn't work for OpenGL on Windows (but, does support OpenGL on Linux).

Instead we will use another tool called RenderDoc, which isn't particularly good for performance, but is handy for debugging OpenGL code. It also works on both Windows and Linux. The Windows version is on Blackboard and the Liunx version can be downloaded from https://renderdoc.org/ . Download and install RenderDoc on your laptop so you can use it in the next step.



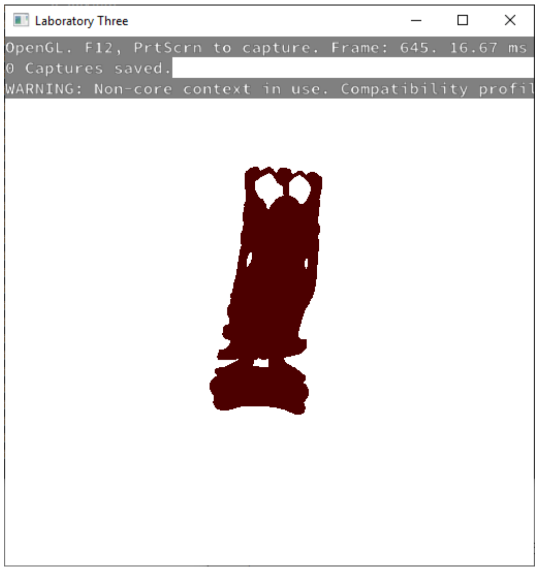
Executable Path: D:\Courses\CSCI 4110\Labs\Lab3\viewer\x64\Debug\viewer.exe
Working Directory: D:\Courses\CSCI 4110\Labs\Lab3\viewer\x64\Debug
Command-line Arguments:
Environment Variables:

[✓] Allow Fullscreen
[✓] Allow VSync
0 secs Debugger Delay
[ ] Collect Callstacks
[ ] Only Drawcall stacks
[ ] Enable API Validation
[ ] Capture Child Processes
[ ] Ref all Resources
[ ] Capture all Cmd Lists
[ ] Verify Buffer Access
[ ] Auto Start

[ ] Queue Capture Frame 0 # Frames: 1

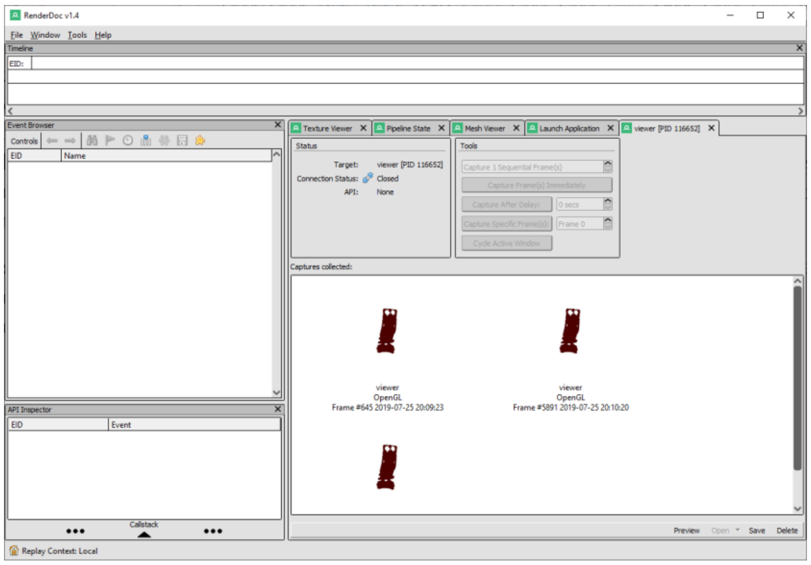
Now start RenderDoc. It should start with the Capture Executable tab as shown in the figure above. If not, just select that tab. In the Executable Path field enter the path to the executable version of your program. The button at the end of the field will bring up a file browser that you can use to select the file. After doing that press the Launch button to start running your program.

The following figure shows what your program should look like when running under RenderDoc. Note the line at the top of the screen that gives FPS and instructions for capturing a frame.



```
OpenGL. F12, PrtScrn to capture. Frame: 645. 16.67 ms
0 Captures saved.
WARNING: Non-core context in use. Compatibility profil
```

To capture a frame press F12 and move the Buddha so your program generates a new frame. It takes some time to capture a frame due to the amount of data involved. Once you have capture several frames you can exit your program and you will be returned to RenderDoc.

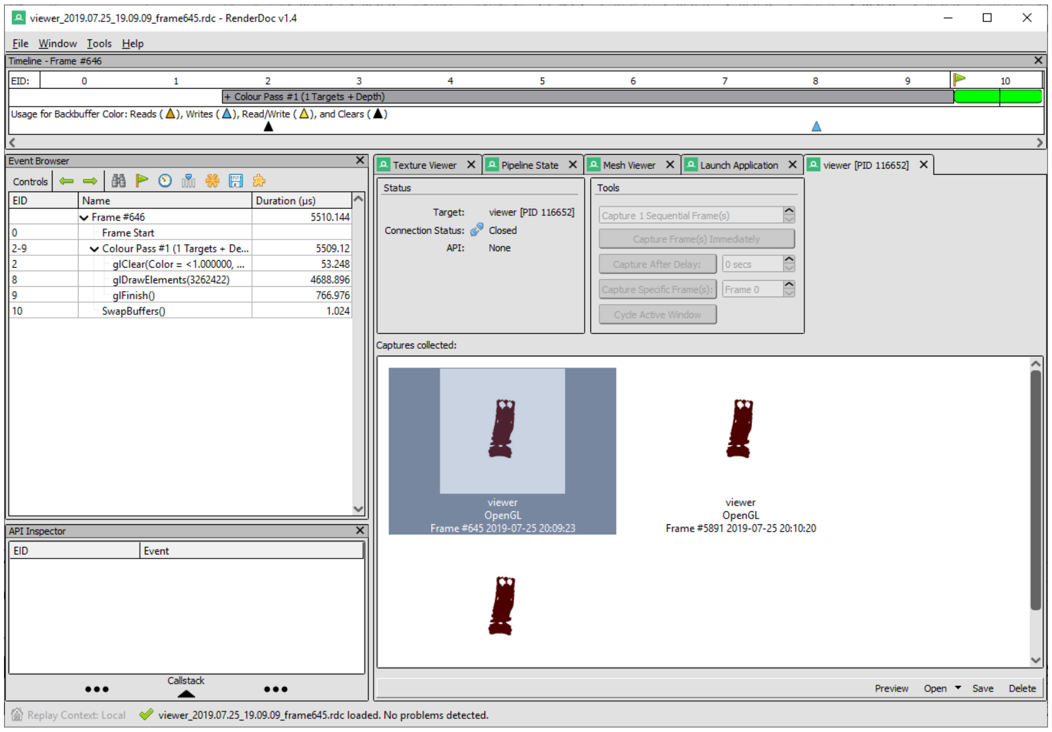


Target: viewer [PID 116652]
Connection Status: Closed
API: None

viewer
OpenGL
Frame #645 2019-07-25 20:09:23

viewer
OpenGL
Frame #5891 2019-07-25 20:10:20

In my case I captured three frames and when I exited the program the three frames are displayed in the Viewer tab. This is shown in the figure above. I double clicked on the first frame to load its data into RenderDoc, this will take a little while since RenderDoc needs to read a fairly large file. The Event Browser panel basically shows us where the problem is, see below. The glDrawElements procedure is sending over 3 million vertex indices to the GPU on each call. This ends up being the bottleneck in our program.



viewer_2019.07.25_19.09.09_frame645.rdc - RenderDoc v1.4

EID	Name	Duration (µs)
	Frame #646	5510.144
0	Frame Start	
2-9	Colour Pass #1 (1 Targets + De...	5509.12
2	glClear(Color = <1.000000, ...	53.248
8	glDrawElements(3262422)	4688.896
9	glFinish()	766.976
10	SwapBuffers()	1.024

Target: viewer [PID 116652]
Connection Status: Closed
API: None

viewer
OpenGL
Frame #645 2019-07-25 20:09:23

viewer
OpenGL
Frame #5891 2019-07-25 20:10:20

viewer_2019.07.25_19.09.09_frame645.rdc loaded. No problems detected.

Spend some time exploring RenderDoc's features. For example look at the Pipeline State tab. From here you can examine the shader code that is used along with their uniform variables. You will find RenderDoc to be useful for debugging OpenGL and Vulkan programs.

Laboratory Report

Show the TA your completed program and the performance results. Also show your TA what you have learned about RenderDoc.