REPORT ON ASSIGNMENT ASPECTS

Alyssa Biasi (s3328976) & Christopher Stojanovic (s3334231)

The first assignment for Real Time Rendering in 2012 investigates the use of 3 methods for rendering on screen objects using OpenGL, GLUT and SDL. In this report, we comment on the effects of these 3 methods (calculating required vertices each frame, henceforth referred to as "immediate mode", storing the data in vertex arrays and finally in vertex buffer objects). We also investigate effects of other variables, such as increasing the tessellation of objects, the number of objects drawn, increasing lighting, etc. We also link these effects to the graphics pipeline and the way the program was designed where appropriate.

Before we go into detail, some benchmarks. This program takes (using a sphere with 1 light source, back face culling disabled);

- Roughly 727ms to render 1,000,000 polygons (tri's)
- Can handle approximately 1152 polygons (tri's) to maintain 30 FPS

Section 1 - Storage Mode

As mentioned above, 3 storage methods have been used for drawing objects to the screen in this assignment - immediate mode, vertex arrays and vertex buffer objects (VBO's). In the following section, we investigate the effect these methods have on the frame time of the program (that is, the amount of time taken to render each frame).

In our first example, we use a sphere shaped object; doubling its tessellation on each data point (note in this case "tessellation" refers to the number of "slices" used to render the object. For a sphere, a tessellation of 8 refers to 8 * 16 tri's, that is, 128 polygons).

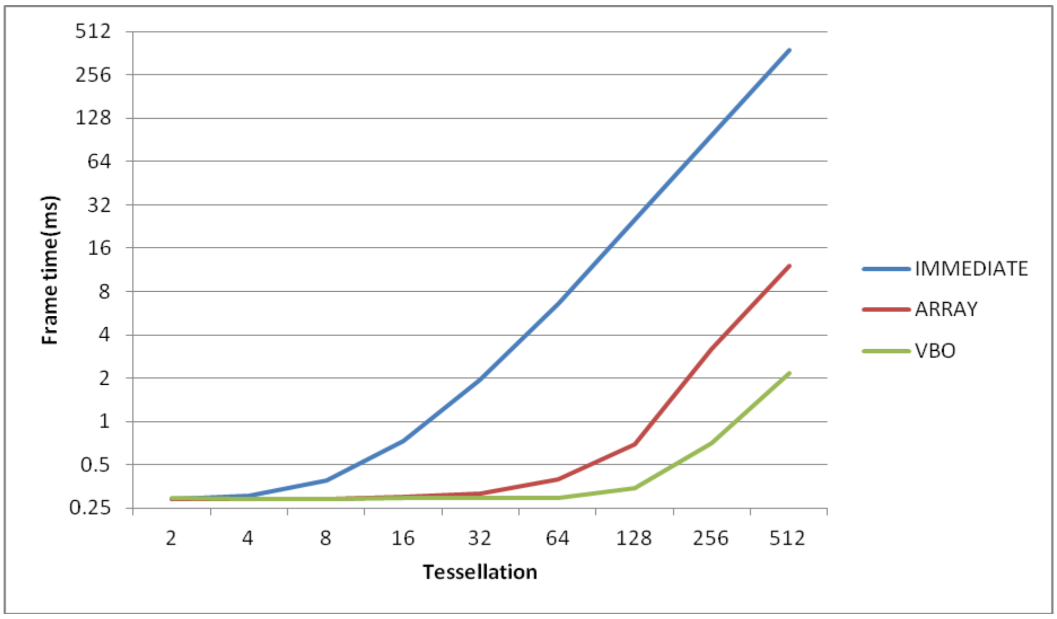

Figure 1 - Tesselation vs Frame Time for a sphere

While the difference between all 3 methods is trivial up to a tessellation count of around 16, we can clearly see the difference between the methods as the tessellation count increases to higher values. Conversely, immediate mode takes at least 16 times longer to render than a vertex array at a tesselation count of 64. The differences are not as significant for vertex arrays and VBO's (especially at lower values), but their differences become more noticible at the extreme ends. The obvious question now is, why is this the case?

The answer comes in the graphics pipeline. Figure 2 outlines a simple example of the graphics pipeline (sourced from the Interactive 3D lecture notes, RMIT 2012).

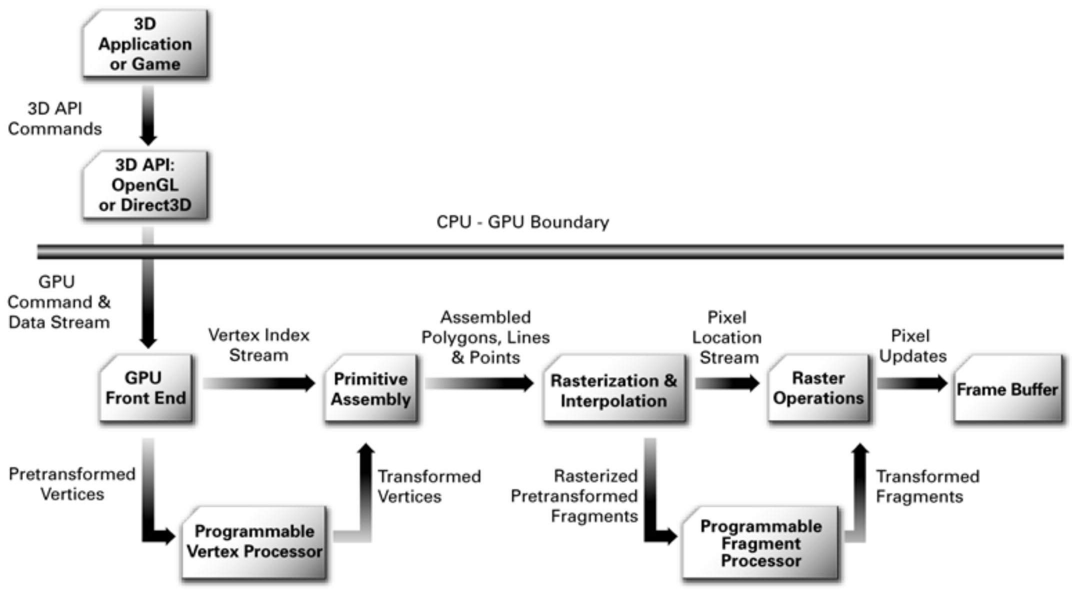

Figure 2 - The graphics pipeline

One of the major causes of slowdowns in a graphics program is "bottlenecks" in this pipeline - in the case of immediate mode, the bottleneck occurs at the CPU/GPU boundary. In immediate mode, the equations for all the vertices are calculated every frame, a task performed on the CPU. This of course means that in every frame new data must be passed down the pipeline into the GPU, slowing performance. Both vertex arrays and VBO's help overcome this be reducing the number of function calls and recycling vertices where possible and the result of this is obvious. VBO's go one step further by storing the data in GPU memory rather than within CPU memory and once again, the impact of this is clear.

Section 2 - Number of objects being rendered

In the following section, we investigate the effect of having multiple objects being rendered on the screen at the same time. Again, all 3 storage methods are used for comparison. A singly lit sphere has been used with its tessellation count at 16 (this value has been chosen as this the point at which the tessellation is high enough for the object to properly resemble a sphere). Grid size refers to the number of objects in the grid (so, a 4x4 grid would contain 16 spheres).

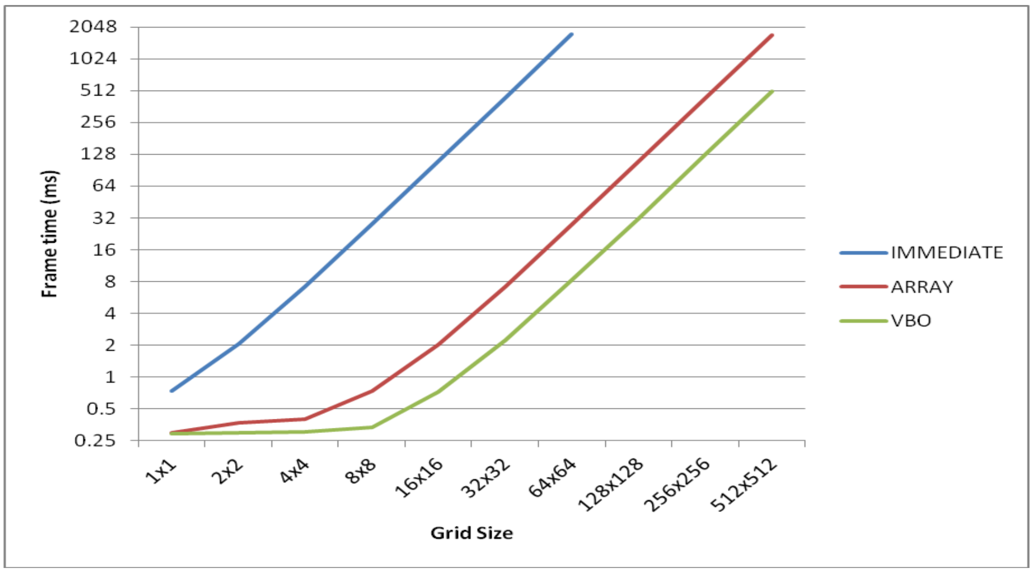

Figure 3 - Grid size vs Frame Time for a sphere

As may be expected, increasing the number of objects being drawn drastically increases the amount of time taken to render each frame (we also again get to see the power of VBO's over vertex arrays and immediate mode). The lack of data points after 64x64 for immediate mode is intentional - the program could not cope with this much information in that state.

As more objects get rendered, we can clearly see a bottleneck is forming in the pipeline as the program struggles to cope with the sheer volume of information. Referring again to Figure 2, it's likely the bottleneck this time is occurring near the primitive assembly as the program tries to organise millions upon millions of vertices.

Section 3 - Object zoom and adjusting window size

We now investigate what effect (if any) zooming in and out on an object has to performance, as well as adjusting the window size. A sphere with the same traits as the one mentioned in section 2 was used. The default zoom of the sphere was used when testing the window size. In Figure 5, the data on the horizontal axis refers to the following;

- Extra small - First appearance of sphere on screen, or smallest possible window size
- Small - Roughly half of the default sphere and screen sizes
- Medium - Size of sphere and screen on load
- Large - Tips of sphere touching the edges of the window. For window size, roughly 150% of the default size
- Very Large - Full screen

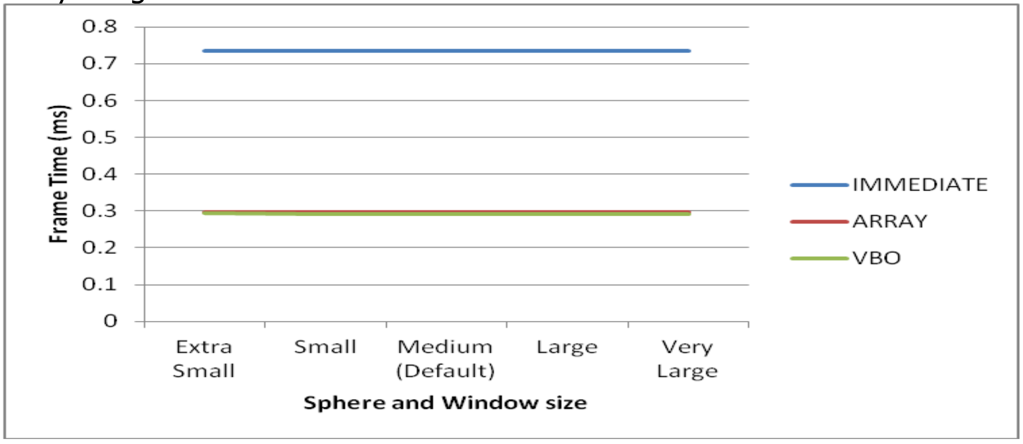

Figure 4 - Sphere and Window size for a sphere

Figure 4 is used to represent both adjusting sphere size and window size. In each of these cases, there is no difference to the frame time. The reason for this is simple: The same information is sent to the GPU regardless of zoom or window size. The program does not bother to check if something is out of view of the "camera". As such, the same amount of processing is done each frame, meaning there is no performance impact from zooming in or out on the object or adjusting the size of the window.

Section 4 - Draw mode (Wireframe/Flat shading/Smooth shading)

By default, our program shades smoothly. However, there are also options to render with flat shading and using a wireframe mode. We now investigate what effects this has on performance. A sphere with tessellation count 16 in immediate mode is used.

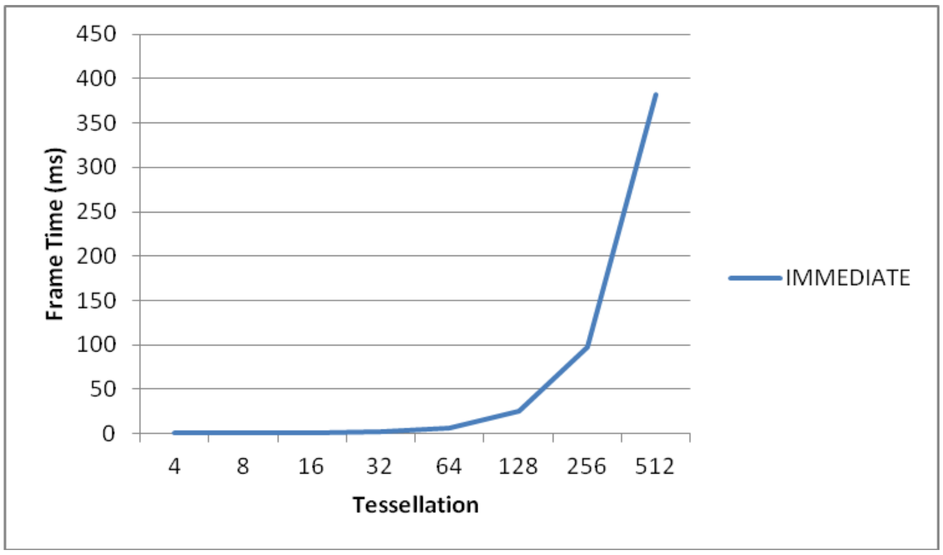

Figure 5 - Tessellation vs Frame Time for a wire frame sphere

When compared with Figure 1, we can see changing to wire frame mode has only a slight change in performance.

The same test is performed for flat shading. The difference between this and the data shown is so small, it does not appear different on a graph. Instead, a table with the different values is shown:

TESSELLATION	WIREFRAME(MS)	FLAT(MS)
4	0.308	0.305
8	0.391	0.39
16	0.732	0.732
32	1.955	1.968
64	6.579	6.636
128	25.1	25.3
256	97	98
512	382.333	385.333

Section 5 - Backface culling

Our next test attempts to see if there is any impact on enabling backface culling has any impact on performance. The same test was performed as in Section 4 and the results mirror that of Figure 5. From this, we can conclude that in our program, backface culling has negligable effect on performance.

Section 6 - Lighting

Finally, we see what effect lighting has on the performance of our program. We again use a sphere at tessellation 16 and use all 3 storage modes for comparison.

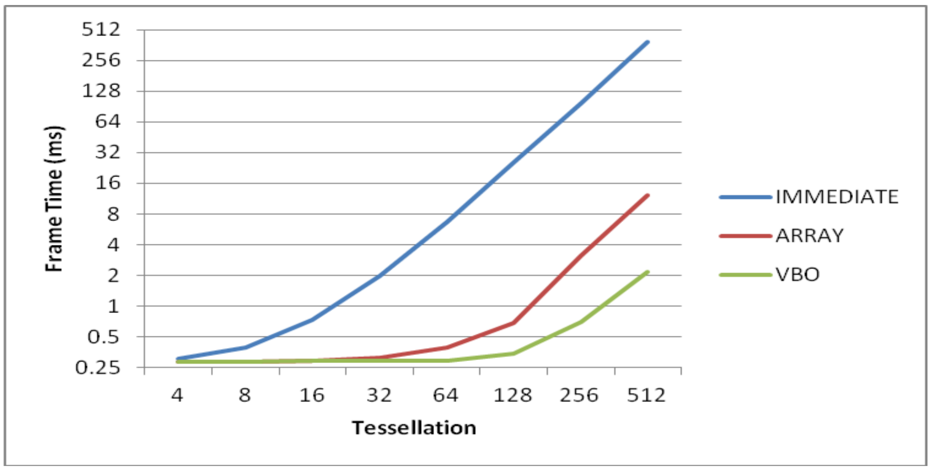

Figure 6 - Tessellation vs Frame Time with lighting off

Clearly, this is not a large performance change. The most logical reason for this is that normals for the sphere are still being calculated, despite not being lit up. If this were to be changed, a perfomance increase would probably be noticed (this was unable to be tested due to time constraints).

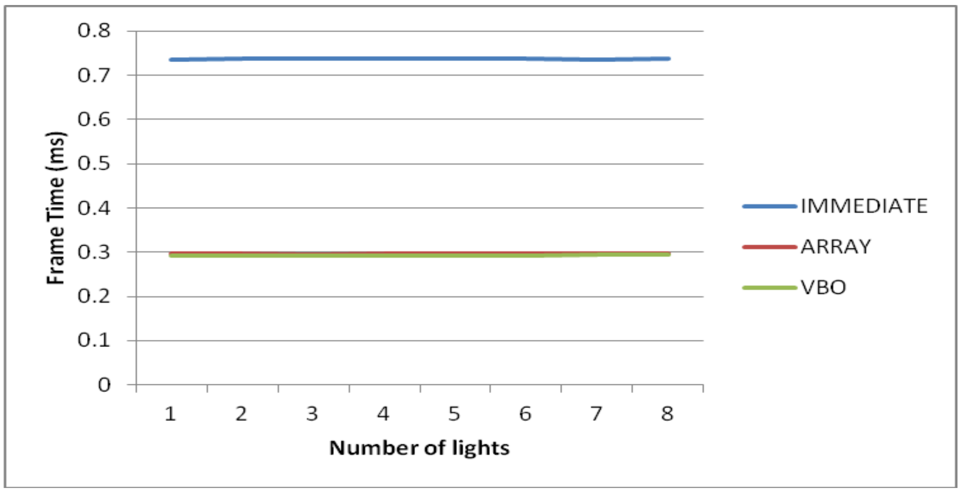

Figure 7 - Number of lights vs Frame Time for a sphere

In our program, adding more lights has a negligiable impact. Our best guess as to why this is happening is simply that the ambient lights are not causing enough of a change to have a significant impact on performance.

Conclusion

From this data, we see using storage modes that use less function calls and which are closer to GPU memory allow for more effiecnt programs. Increasing the number of polygons to draw also has a signficant impact on program performance, no matter what the storage mode. Furthermore, various other factors such as lighting and render modes can have varying effects on performance often linked to the implementation of them in the program.