Real Time Rendering

Assignment 1 - Report

20/08/10

Tom Harris s3236050
Alon Gal s3216299

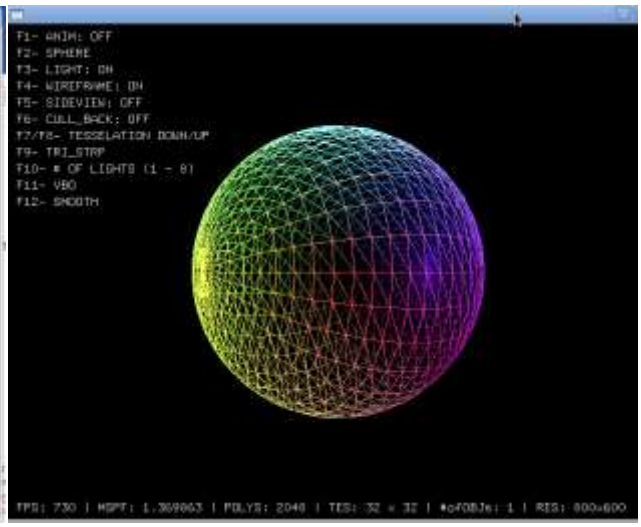
Introduction

The aim of this assignment was to investigate the graphics performance of procedurally generated objects using advanced rendering techniques.
To achieve this we implemented a C based SDL/OpenGL application that allowed rending options to be altered at run time with interactive controls.
The frame rate, poly count and time per frame achieved by each rendering mode was displayed in the applications window so that the rendering techniques could be compared. The tests where run on the on the machines in the Sutherland lab.

A list of the run time options:

Drawing mode:	Triangles/Triangle Strips/ Quad Strips
Rendering mode:	Display Lists/ Vertex Arrays/Vertex Buffer Objects/Immediate
Geometry:	Torus/Sphere
Tessellation:	Double/Half (min 4x4)
Back face culling:	On/Off
Number of objects:	Increased/Decreased (Displayed in a grid)
Camera View:	Side/Front
Camera Zooming:	In/Out
Animation:	On/Off (Objects rotate about vertical access)
Lighting:	On/Off
Number of lights:	1 – 8 (Can be increased and decreased)
Wireframe:	On/Off
Shaded:	Smooth/Flat

*Keys for the main rendering controls are visible in the application window. A complete list of control keys is available in the readme.



General Testing

Test 1

As a base for our investigation, we recorded the results achieved with no optimisation. This meant having each option set to its worst performing state.
We collected results over a range of tessellations.

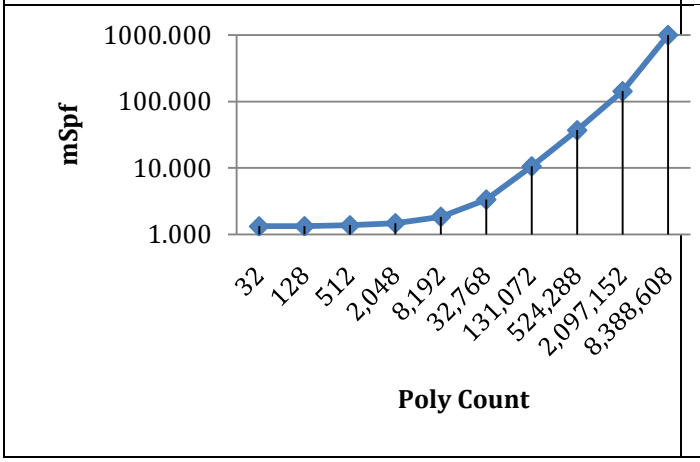
Test 2

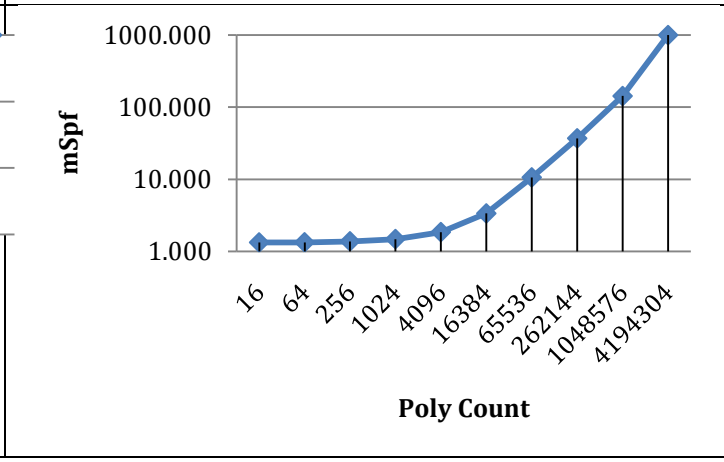
Then with a benchmark in place we ran the same tests again but with the best performing states of each option.

Test 1		**Test 2**	
Drawing mode:	Triangles	Drawing mode:	Triangle Strip
Rendering mode:	Immediate	Rendering mode:	VBO
Geometry:	Sphere	Geometry:	Torus
Back face culling:	Off	Back face culling:	On
Number of objects:	1	Number of objects:	1
Camera View:	Front	Camera View:	Side
Camera Zooming:	Stationary	Camera Zooming:	Stationary
Animation:	On	Animation:	Off
Lighting:	On	Lighting:	Off
Number of lights:	8	Number of lights:	(lights off)
Wireframe:	On	Wireframe:	Off
Shaded:	Smooth	Shaded:	Flat

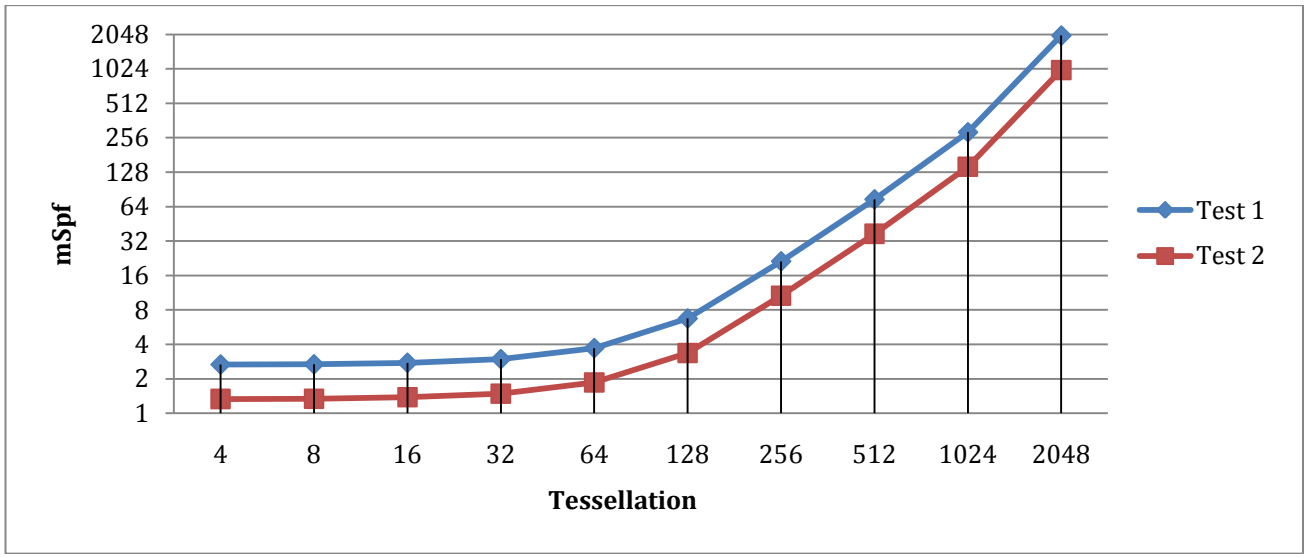
Tess	FPS	mSPF	Polys
4x4	752	1.333	32
8x8	747	1.338	128
16x16	724	1.381	512
32x32	674	1.485	2048
64x64	539	1.855	8192
128x128	295	3.367	32768
256x256	95	10.638	131072
512x512	27	37.037	524288
1024x1024	7	142.857	2097152
2048x2048	1	1000.000	8388608
4096x4096	0	inf	33554432

Tess	FPS	mSPF	Polys
4x4	755	1.326	16
8x8	747	1.338	64
16x16	745	1.340	256
32x32	739	1.351	1024
64x64	724	1.382	4096
128x128	663	1.510	16384
256x256	450	2.217	65536
512x512	169	5.882	262144
1024x1024	50	19.607	1048576
2048x2048	13	76.923	4194304
4096x4096	3	333.333	16777216





Test 1 vs. Test 2
Based on Tessellation (mSpf Logarithmic Scale)



Having collected the above data, we could see the difference between the "best" and "worst" possible cases for our testing application.
We noticed that there was a very large difference in performance as the tessellation was increased.
The next step was to identify which rendering options contributed the most and least to this change in performance. This was achieved by individual option testing.

Animation
Turning on animation had very little effect on the applications performance. This could be due to our animations simplicity.

Triangles/Triangle Strip/Quad Strip
There was not much difference between drawing with Triangle strips and Quad strips. We expected these results because both drawing methods involve the same number of openGL calls. Drawing with plain Triangles however caused a large decrease in performance. This is not surprising due to the extra/redundant openGL calls required to use this drawing mode.

Lighting (any number, any colour)
A surprising result, turning on eight lights had no effect on performance. This is very interesting result as we had expected lighting calculations to be a heavy task. This result implies the simplicity of the standard openGL lighting calculations. These results also could be due to the optimisation of floating point calculations on modern GPU's.

Flat vs Smooth
Toggling between flat and smooth shading mode had a negligible effect on the time per frame achieved by the application. This was also surprising considering the large affect this has on the appearance of the object. Perhaps another example of the GPU's optimised floating point calculations.

Back face Culling
Turning culling on had a large effect on performance when applied to Quad and Triangle Strips, this is no surprise as it reduces the amount of polygons to be rendered.
What was surprising was that the boost was only small when drawing using Triangles.

Wireframe vs Fill
Surprisingly wireframe performed worse then fill mode in all tests, this is not the result we had expected. We had thought the interpolation of the "fill" colours and the increased number of pixels being rendered would result in reduced performance. Maybe these fill calculations have been optimised to go fast, since most of the time people wish to view their graphics is in a filled mode.

Tessellation and Grid Size
Since increasing and decreasing the tessellation of the objects or the number of objects ultimately increase the number of polygons to be rendered there is and obvious impact on performance.

Torus vs Sphere
We expected toggling between the torus and sphere would have very little effect on the frame rate, due to the similarities draw functions. We were correct, although the torus did occasionally outperform the sphere by a small margin, this may be due to the "hole" in the torus, meaning less rendering is required.

Side View
Changing to side view, even with a grid of objects, only provided a small boost in performance. This maybe because the camera was in perspective mode and therefore the objects only partially hide each other when they were in a line.

Rendering mode (Display Lists/ Vertex Arrays/Vertex Buffer Objects/Immediate)
When we trailed the different rendering modes, we found that they had a very large effect on the performance of the application. We further investigated their effects, tests and analysis below.

Render Mode Specific Testing

With a general analysis complete we looked at the performance of the advanced rendering techniques relative to each other.
With all other rendering options static, we compared Vertex Arrays Display Lists and Vertex Buffer Objects against Immediate mode.

We ran two tests, both with the **same number of polygons displayed**.
The first test had many objects, with a low tessellation.
The second test had a high tessellation with only one object.

The following options where used:

Test 1: (low tess, many objects)
Number of objects: 4,096
Tessellation: 32
Polys: 4,194,304

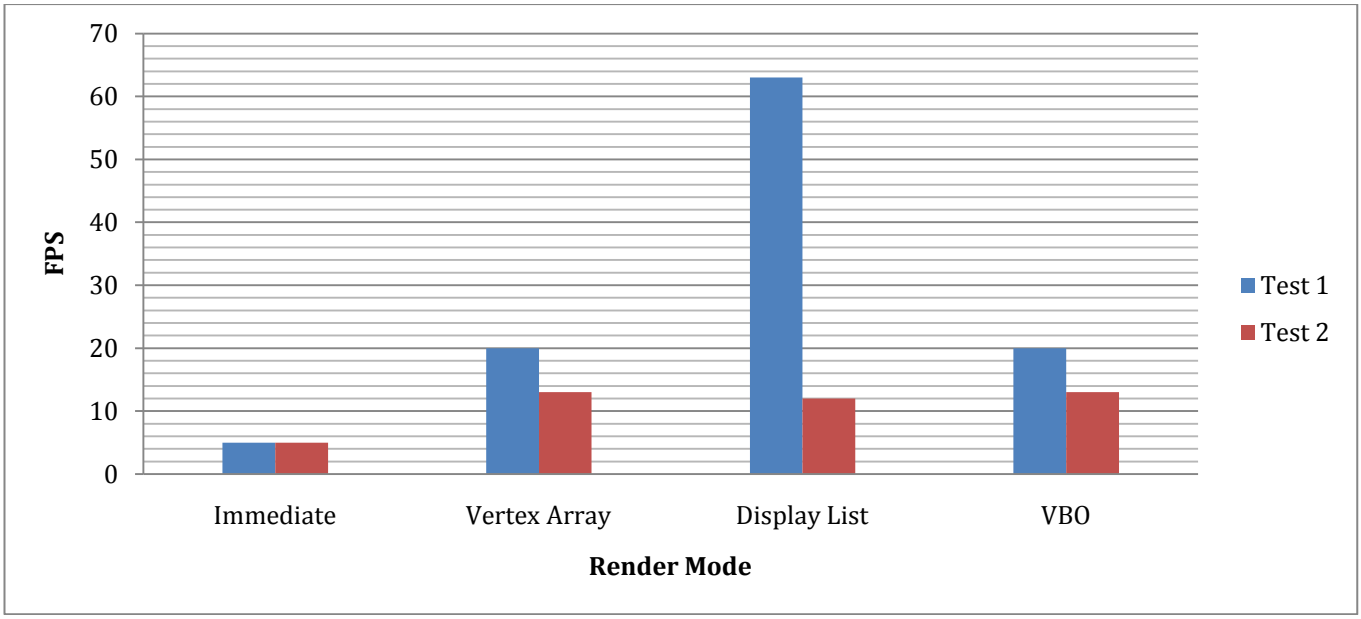
Test 1	FPS	mSPF
Immediate	5	200.000
Vertex Array	20	50.000
Display List	63	15.873
VBO	20	50.000

Test 2: (high tess, 1 object)
Number of objects: 1
Tessellation: 2,048
Polys: 4,194,304

Test 2	FPS	mSPF
Immediate	5	200.000
Vertex Array	13	76.923
Display List	12	83.333
VBO	13	76.923

Other Options (Test 1 and Test 2)

Drawing mode:	Triangle Strip	Camera View:	Front
Rendering mode:	Variable	Geometry:	Torus
Camera Zooming:	Stationary	Back face culling:	On
Animation:	Off	Lighting:	On
Wireframe:	Off	Number of lights:	8
Shaded:	Smooth		



Immediate
No surprise that immediate mode performed poorly in both tests, this technique represents the most basic of the four rendering options.

Vertex Array
We expected the vertex arrays to give a significant boost in performance and they did, this was expected because we are delivering the data to the GPU in a much more organised form and reducing the amount of redundant data being sent.

Display List
Surprisingly display lists gave a very different result in the two tests, while they provided less benefit then VBOs and Vertex Arrays for 1 object with a high tessellation, they blitzed the competition when the same object with a low tessellation is drawn many times.
This implies that the best time to use display lists is when you are drawing the same geometry many times, this allows the GPU to take maximum advantage of the static pre compiled code that is a display list.

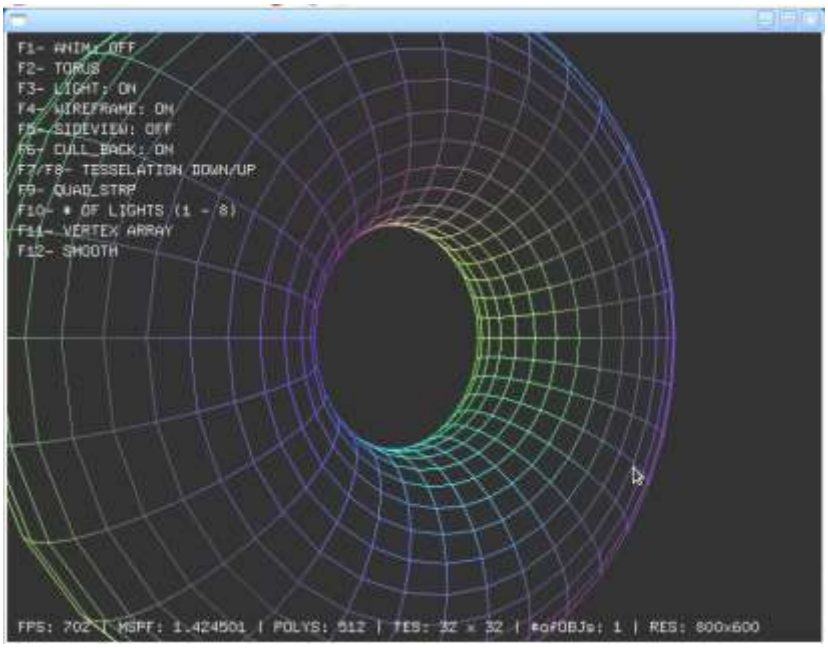
Vertex Buffer Objects
A surprising result, we expected VBO's to be the outright best performer, although they consistently performed equally with vertex arrays. We tried using the static and dynamic draw options although this did not affect the performance. It may have been our implementation of VBO's which caused this result. We were binding and unbinding for each object to be drawn (please refer to code), this could have been creating a bottleneck as it is an expensive operation. We were also using separate arrays for Normal's, Indices and Vertices, better performance may have been achieved if we had used an interleaved array.

Other Testing

30 FPS
How many lit and shaded polygons can we render to achieve exactly 30 frames per second?
With base settings (Immediate mode and glTriangles) we rendered 528,392 polygons. We used a single object with a tessellation of 514.

We did achieve much higher poly counts at *approximately* 30 frames per second using VBOs and

Triangle Strips although we were not able to achieve an exact 30 fps boundary using these techniques. A better example of the potential of the system would be that we were able to render 1,048,576 polys at a whopping 50 frames per second using VBOs and Triangle Strips, although we did not achieve exactly 30 fps.



<u>1 Million Polys</u>
It took 19 milliseconds to render a single lit object with 1,048,576 polygons.
To achieve this we used VBO's and Triangle Strips.

<u>Resolution (increasing window size and zooming)</u>
All of the above testing was run at a resolution of 800x600, we ran some of the same tests again and found that increasing the resolution to 1024x786 and zooming did not have much effect on the performance of the application.
At a tessellation of 256, rendering using VBO's and the higher resolution, we noticed a loss in performance by 5 frames per second. At higher tessellations, the results were the same. At lower tessellations, no consistent loss of performance was seen.