EEC 277- Assignment 2
Characterization/Reverse-engineering Assignment

By- Peiyao Shi and Prashant Gupta

The purpose of this report is to explore characteristics and performance analysis on GPU. For this assignment, we have picked two different graphics hardware, namely **Intel HD Graphics 5000** and **NVIDIA GeForce GT 650 M**. **Part 1** of the report covers how variation of different parameters affect GPU performance while **Part 2** talks and compare different anti-aliasing techniques.

Part 1:

For the entire experimental analysis units of data considered are as below-

1. Area of triangle- In pixels
2. Triangle Rate – Million triangles/second
3. Geometry Rate- Million Vertices/ second
4. Frame Rate- Frames/ second
5. Fragment Rate- Million Fragments/ second

For the analysis of GPU's performance, the impact of following parameters was tested using wesbench. The code used for the analysis can be found here.

a) Triangle Size

As the triangle size increases, the Fill rate starts to increase while the Vertex Rate/Geometry Rate keeps on decreasing. As the triangle area, the number of fragments becomes large enough that most of the GPU computing power is focused on that and thus the geometry rate drops. After the triangle area is large enough both the fill and geometry rate saturates which is a sign of rasterizer getting maxed out.

The crossover point is when in performance of the processor is neither geometry limited nor fill limited.

The crossover point for Intel HD Graphics 5000 = 80 Pixels
The crossover point for NVIDIA GeForce GT 650 M = 28 Pixels

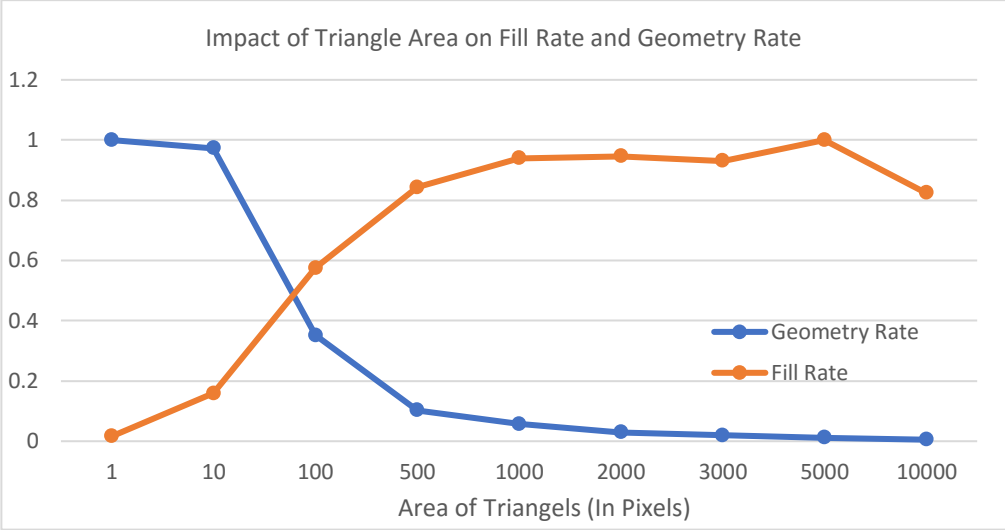

Fig 1: Normalised Graph of Fill and Geometry Rate for Intel HD Graphics 5000

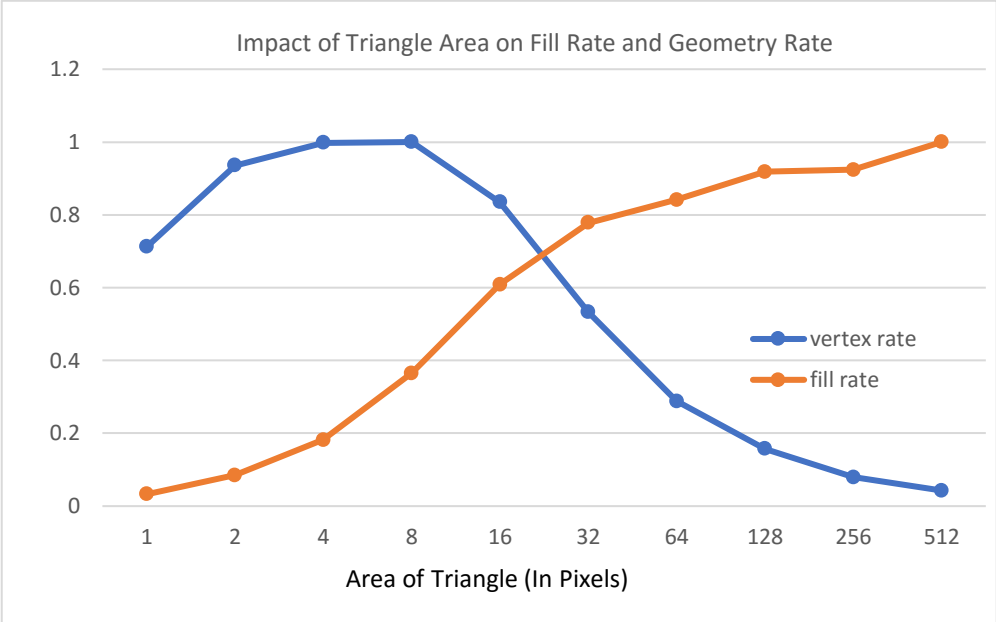

Fig 2: Normalised Graph of Fill and Geometry Rate for NVIDIA GeForce GT 650 M

b)

a. Lightning

It is overserved that turning the lightning on has no effect on Graphics performance. It is possible because of some specialized hardware/optimization in GPU which takes care of rendering with lightning without having any effect on GPU performance. The observation was found to be similar for both the GPU's. Hence, no change in crossover point is observed.

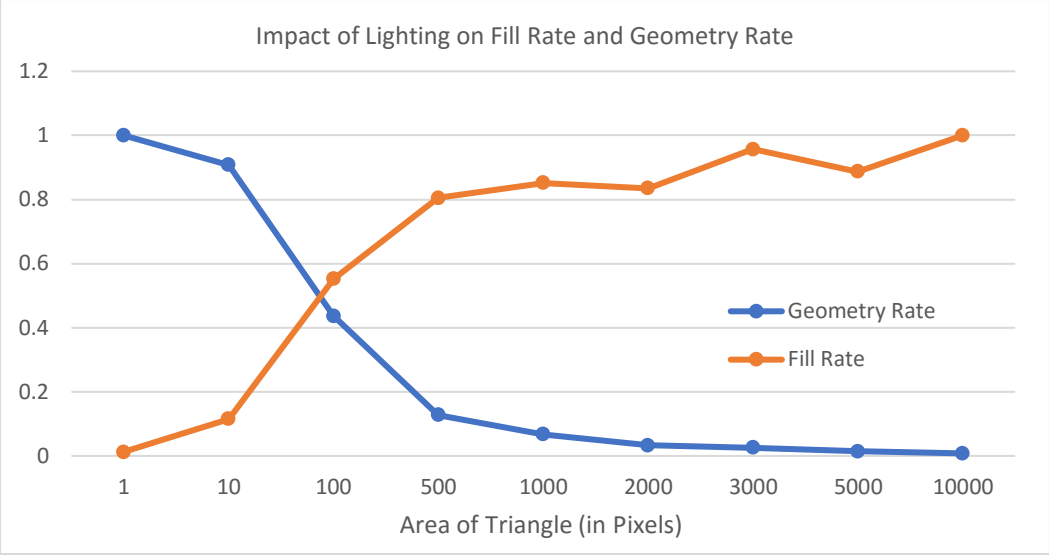

Fig 3: Normalised Graph of Fill and Geometry Rate with lightning for Intel HD Graphics 5000

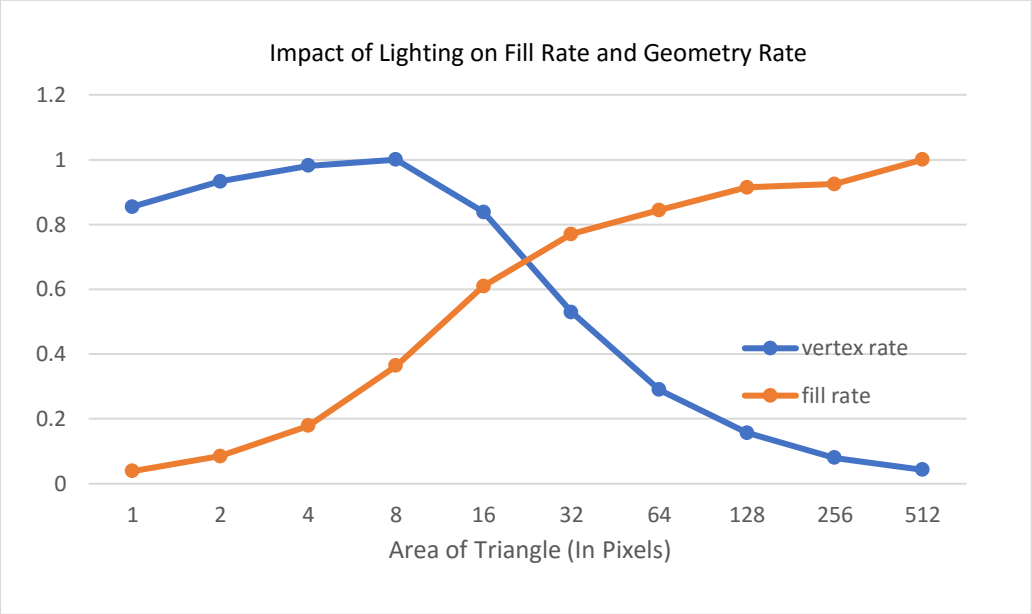

Fig4: Normalised Graph of Fill & Geometry Rate with lightning for NVIDIA GeForce GT 650 M

b. Texturing

As seen from figures 5 and 6, texturing enabled has a significant impact on the Graphics performance. As texturing is a fragment operation it reduces the fill rate significantly but doesn't have any impact on the geometry rate. Hence, the crossover point shifts to the right.

- The crossover point for Intel HD Graphics 5000 with texturing = 105 Pixels
- The crossover point for NVIDIA GeForce GT 650 M = 32 Pixels

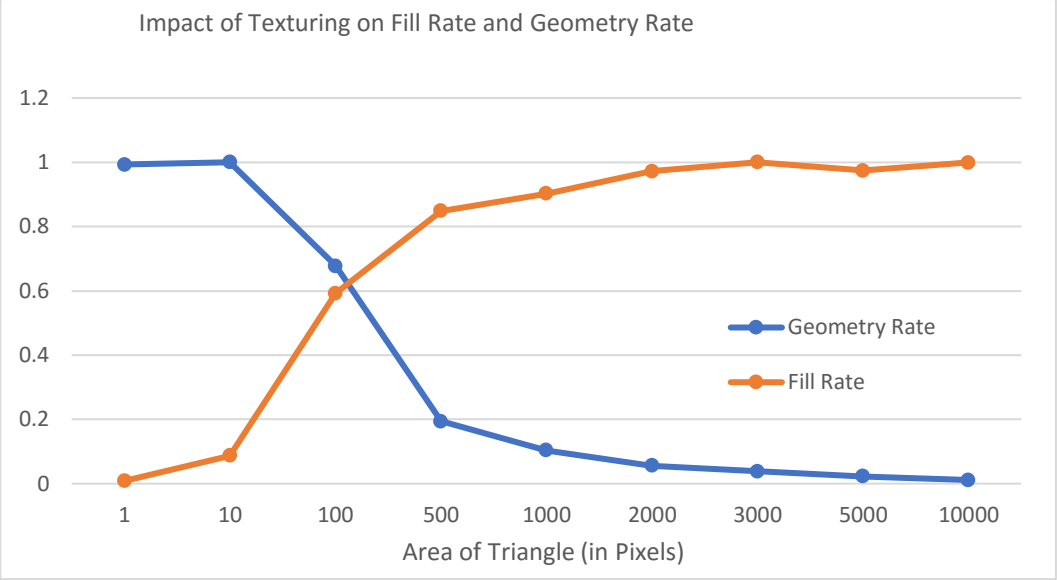

Fig 5: Normalised Graph of Fill and Geometry Rate with texturing for Intel HD Graphics 5000

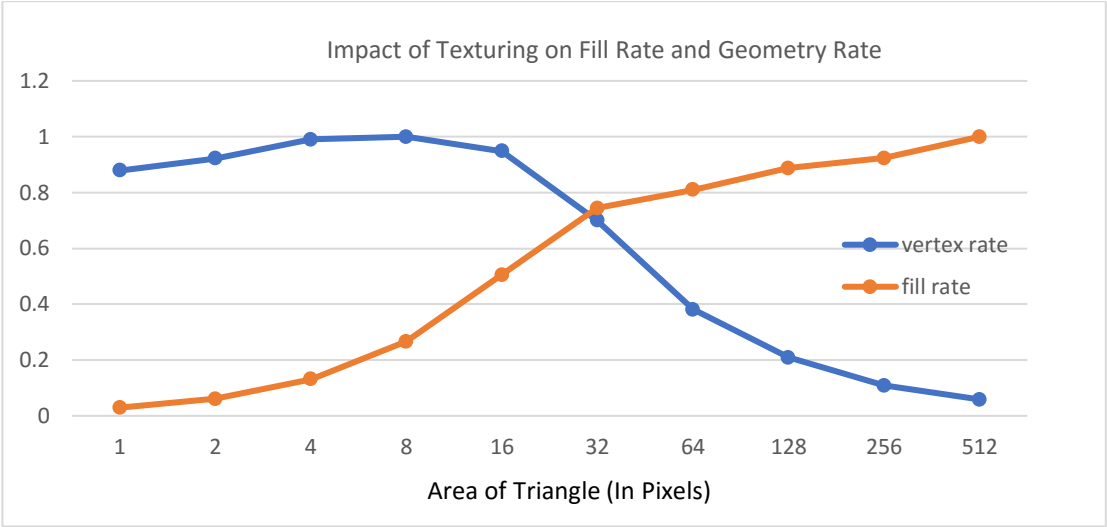

Fig 6: Normalised Graph of Fill & Geometry Rate with texturing for NVIDIA GeForce GT 650 M

c. Lightning and Texture

When both lightning and texture are enabled, the only effect seen is because of texture as we have already seen that lightning does not impact the GPU performance. Hence the crossover points remain same as part b.

The crossover point for Intel HD Graphics 5000 with texturing = 105 Pixels
The crossover point for NVIDIA GeForce GT 650 M = 32 Pixels

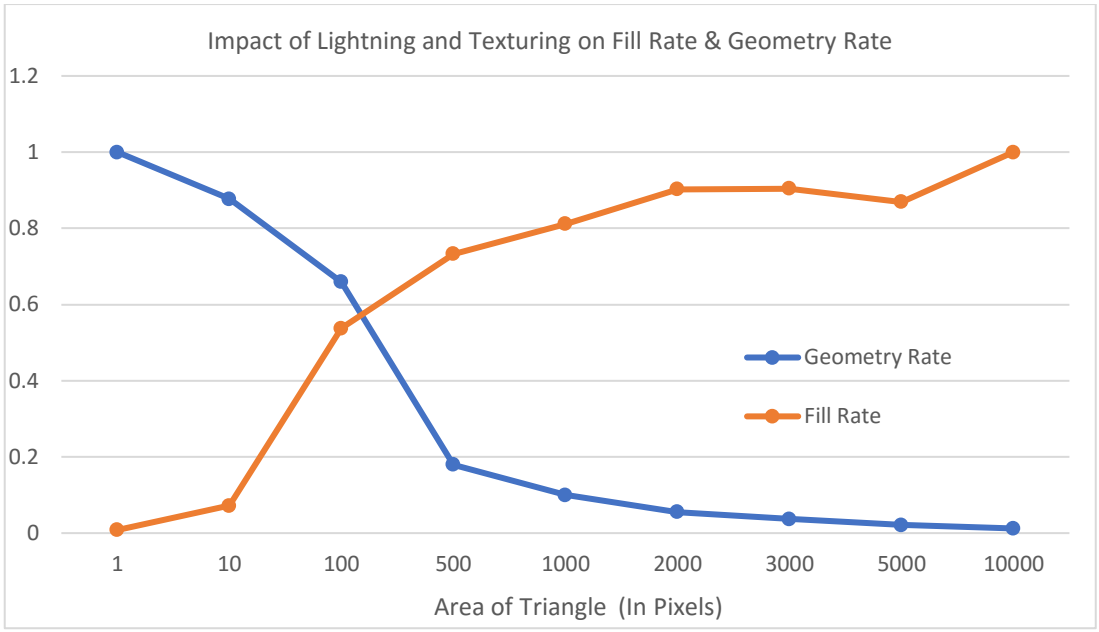

Fig 7: Normalised Graph of Fill and Geometry Rate with texturing & lightning enabled for Intel HD Graphics 5000

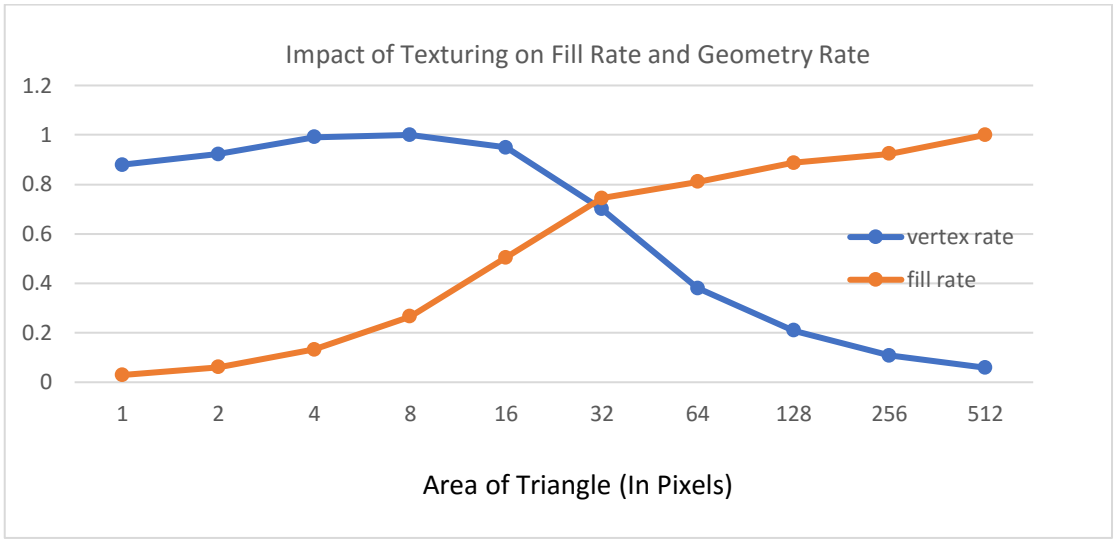

Fig 8: Normalised Graph of Fill & Geometry Rate with texturing & lightning for NVIDIA GeForce GT 650 M

d. Triangle type

The disjoint triangle graphs look exactly like the graph from the first part as it is the default triangle type in wesbench. When analysing triangle strips the performance of both fill rate and geometry rate was seen to improve for both the processors.

- The crossover point for Intel HD Graphics 5000 is 70 pixels with triangle strips while 80 pixels with disjoint triangles
- The crossover point for NVIDIA GeForce GT 650 M is 24 pixels with triangle strips while 28 pixels with disjoint triangles.

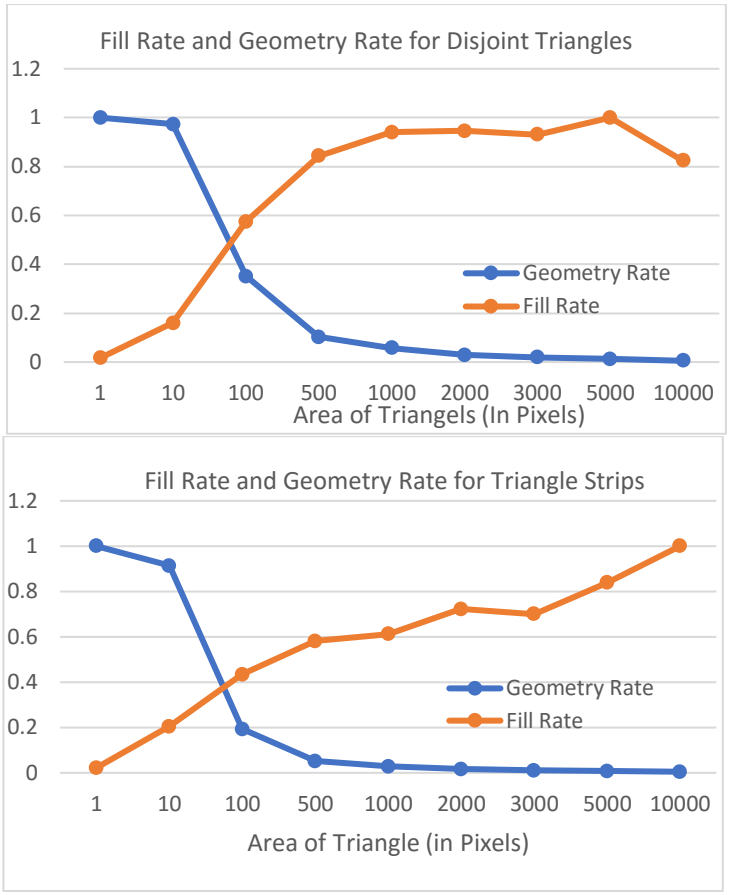

Fig 9: Normalised Graph of Fill and Geometry Rate for Intel HD Graphics 5000

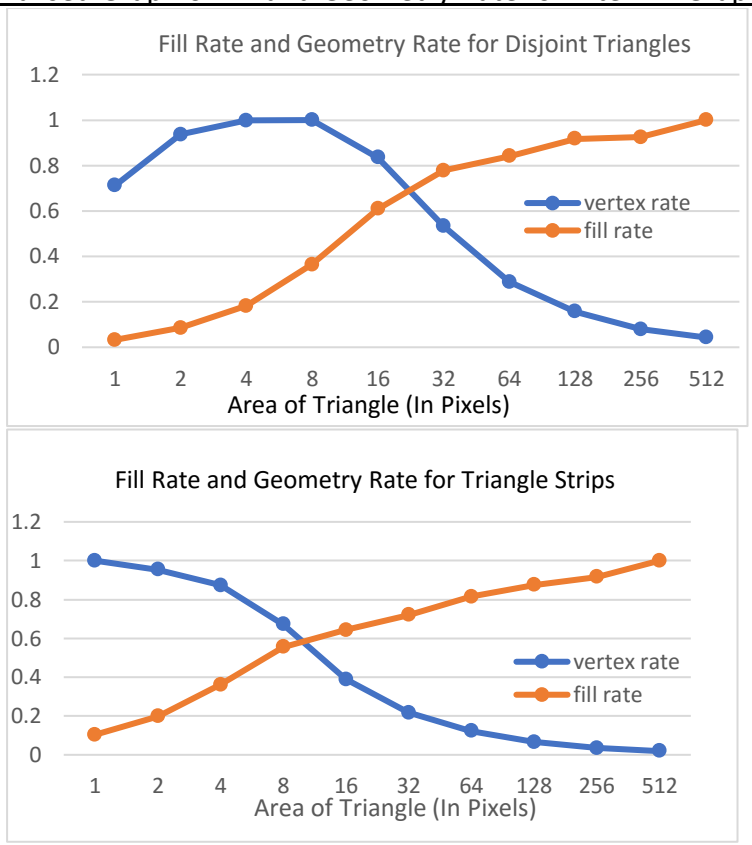

Fig 10: Normalised Graph of Fill and Geometry Rate for NVIDIA GeForce GT 650 M

c) Batch Size

When batch size is small the Fill rate increase with the vertices per bucket, this is because the GPU is limited by CPU's ability to send work. For Intel HD Graphics 5000 the system stops being interface limited at 510 vertices/bucket and starts being GPU limited, for NVIDIA GeForce GT 650 M the value lies at 255 vertices/bucket. Geometry Bandwidth between CPU and GPU is 193.83 Mverts/sec (vertex size is 128 bits; hence bandwidth will be 2.87 GB/s) for Intel HD Graphics 5000 and 244.08 Mverts/sec (3.6 Gb/s) for NVIDIA GeForce GT 650 M.

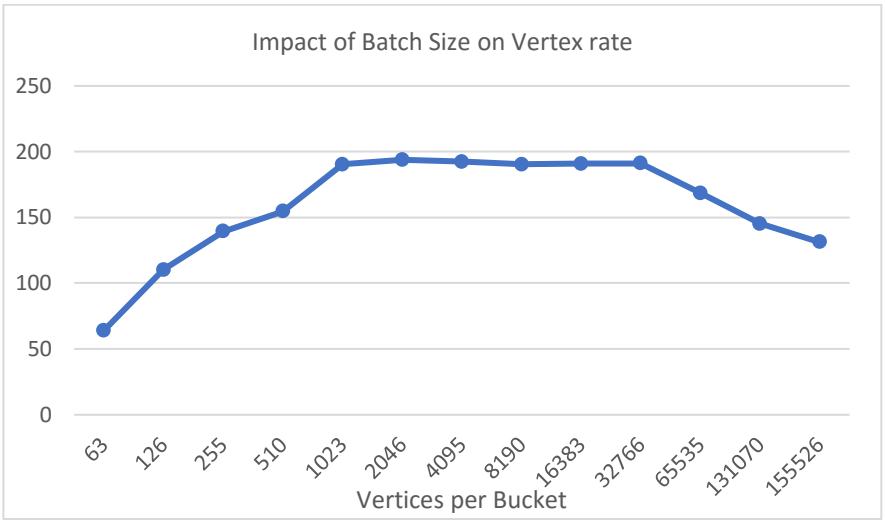

Fig 11: Graph of Fill Rate vs Vertices per Bucket for Intel HD Graphics 5000

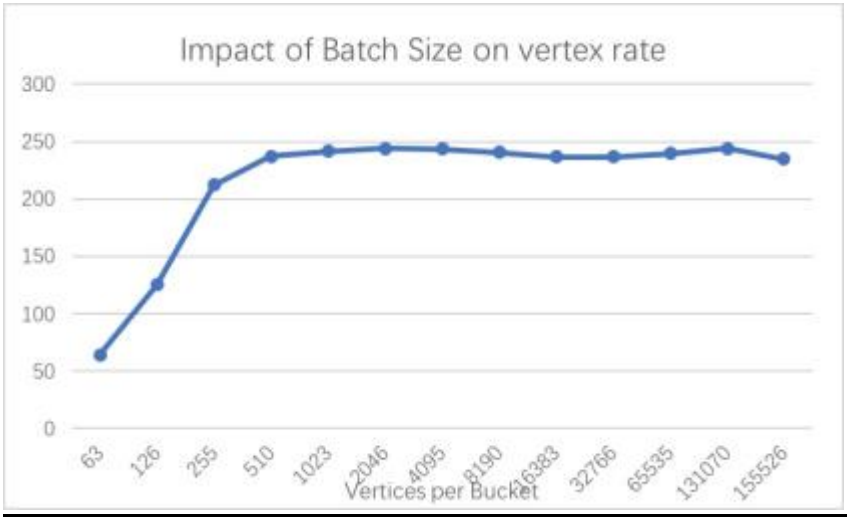

Fig 12: Graph of Fill Rate vs Vertices per Bucket for NVIDIA GeForce GT 650 M

d) Texture size

a. Fill Rate

When the texture size is small because of texture caching we observe a high fill rate, but as the texture size increases the cache cannot hold it. So, every time texturing is required the program must go to main memory to fetch the texture data and hence the performance of fill rate decreases significantly when the texture hits the (512*512) mark in case of Intel HD Graphics 5000 and (2048*2048) in case of NVIDIA GeForce GT 650 M. In this case texture bandwidth, between the CPU and the GPU will be a bottleneck.

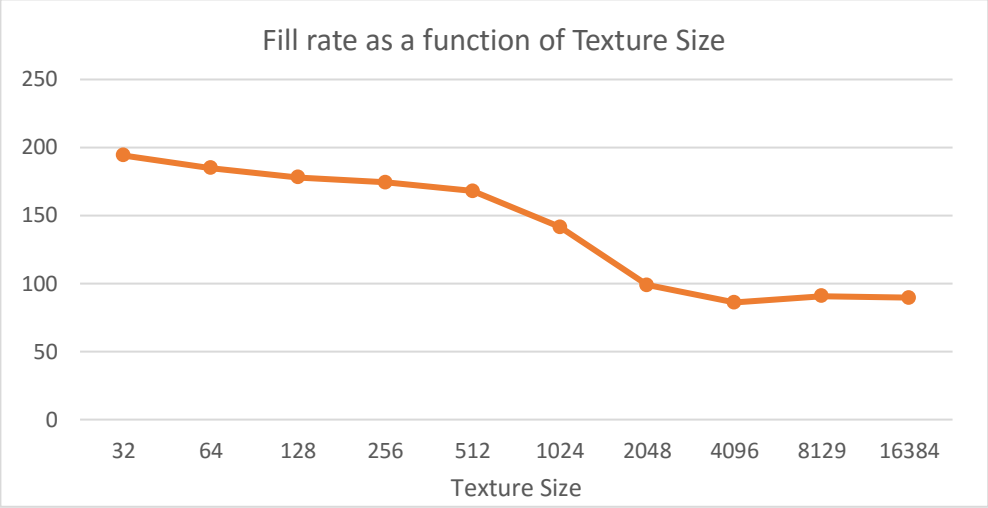

Fig 13: Graph of Fill Rate vs Texture Size for Intel HD Graphics 5000

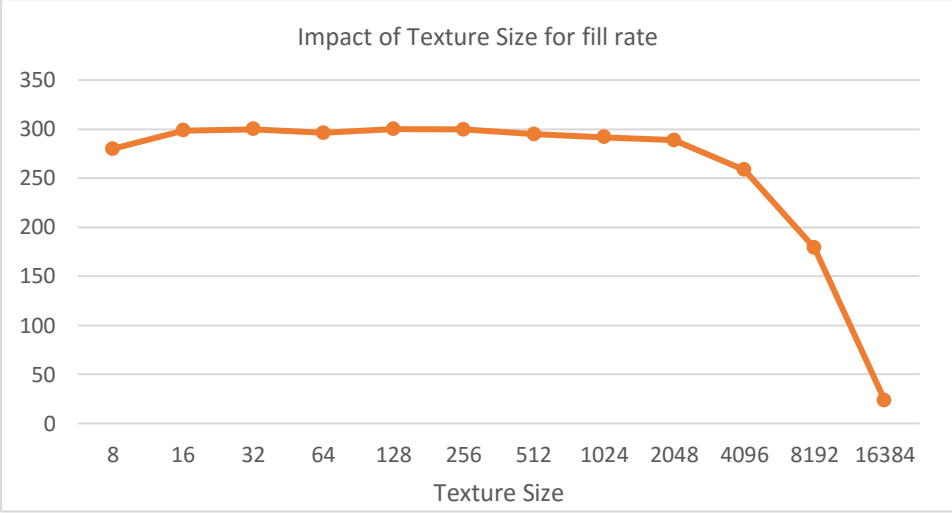

Fig 14: Graph of Fill Rate vs Texture Size for NVIDIA GeForce GT 650 M

b. Texture Bandwidth

In the wesbench program we are using for this experiment, the texture storage mode is RGBA which is 4 bytes long. Using the GL_nearest mode we know that the number of texel per fragment is 4. Hence the texture bandwidth will be – 4 bytes/fragment/second.

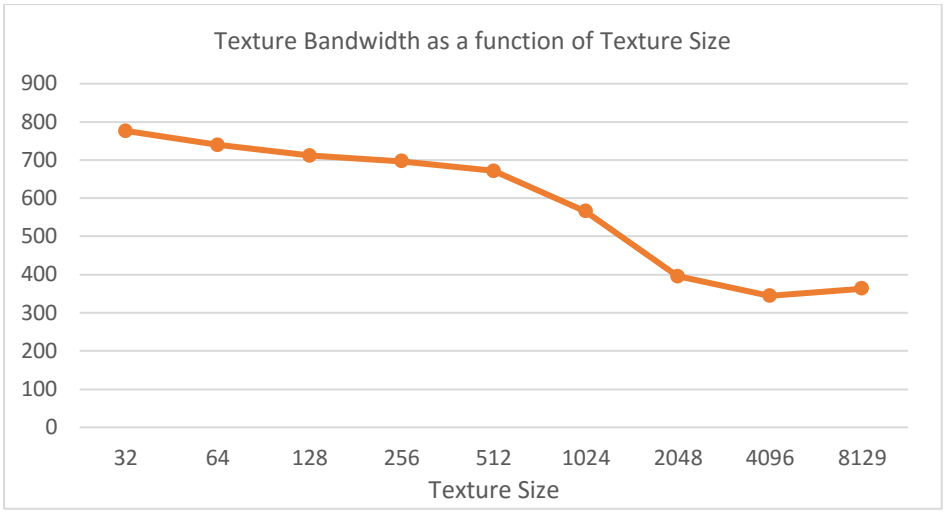

Fig 15: Graph of Texture Bandwidth vs Texture Size for Intel HD Graphics 5000

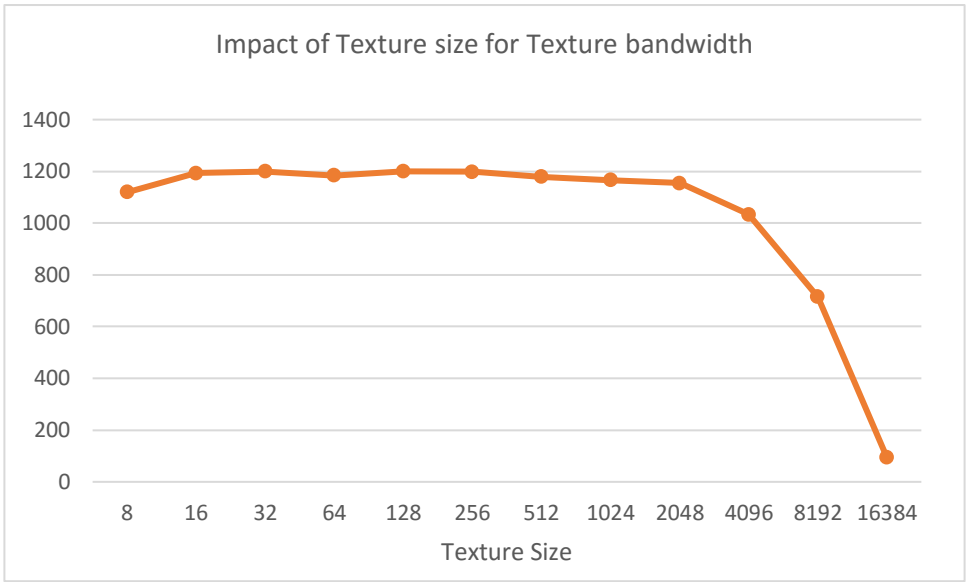

Fig 16: Graph of Texture Bandwidth vs Texture Size for NVIDIA GeForce GT 650 M

Comparing 2 mipmapping techniques – GL_LINEAR with GL_NEAREST

The following graphs shows the comparison between GL_Nearest and GL_Linear mipmapping techniques. For Intel HD Graphics 5000 we can GL_Linear is performing better for small triangles but for large traingles GL_Nearest is performing better while for NVIDIA GeForce GT 650 M the performance of both GL_Linear and GL_Nearest almost remains same. For Intel graphics card, we observe a crossover at a texture size of 2048*2048. We think this trend is happening because of either some hardware accelerator or spatial locality which is causing these trends. For Nvidia graphics card, we believe this is achieved by some specialised hardware.

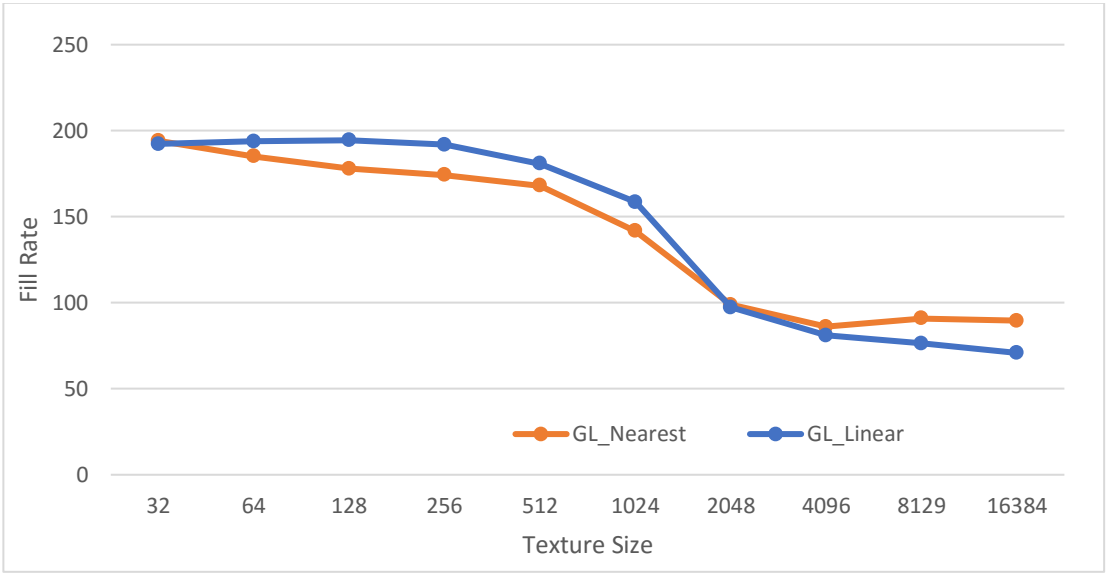

Fig 17: Texture Size V/S Fill Rate graph for GL_Nearest and GL_linear using Intel HD Graphics 5000

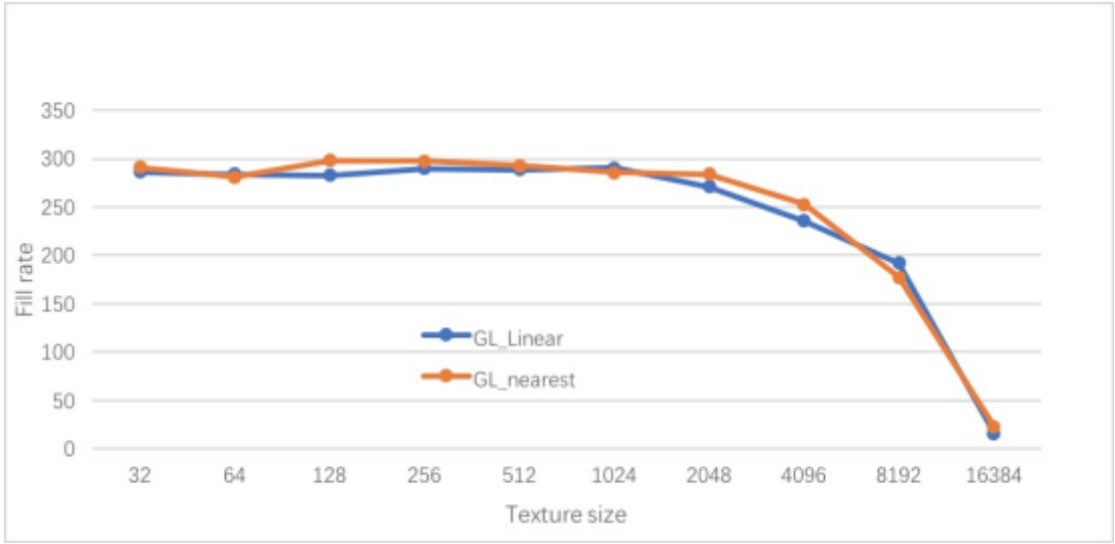

Fig 18: Texture Size V/S Fill Rate graph for GL_Nearest and GL_linear using NVIDIA GeForce GT 650 M

Part 2:

What are the sample locations within a pixel with different sampling modes (different antialiasing levels)? For instance, if you set your graphics pipeline to sample 4 times per pixel, where are the 4 sample positions associated with that pixel? How are those samples combined to yield the final color of a pixel?

By default, there is only one subsample location within a pixel that we expect should be in the centre of each pixel. If that subsample is covered under the geometry, we see filled pixels and if that subsample is not covered, we get a blank pixel. This is too less of a resolution and that is why get pixelated edges when we try to zoom the figure around the edges. Below are the sample we got when we tried to simply make a triangle with OpenGL. The second zoomed in image show the pixelated edges.

Fig 19: With default subsample size as 1, we can see the pixelated images

The solution to this is anti-aliasing where we can increase the number of subsamples to get better resolution (e.g. using multisample anti-aliasing we can increase the number of subsamples). The pixel colour is determined by the no of subsamples covered by the geometry and will enable a smooth transition of color at the edges.

Fig 20: Left image as subsample size of 1 while the right image has subsample size of 8

The above experiment show the mutisample anti-aliasing solving the problem by increasing the number of samples.

1. Supersampling Anti-Aliasing (SSAA)-

SSAA is an old technique in which the color samples are taken multiple times inside a pixel and the average colour is calculated. This is achieved by rendering the image at much higher resolution than the one to be displayed and then shrinking the image to get the desired size. This down-sampling results in smoothening of the edges of the object. In SSAA, the same pixel is shaded multiple times per pixel which results in heavy computation load. Although, this was the one of the earliest techniques of anti-aliasing but because of its inefficiency it has been outdated and it gave way to better techniques like MSAA which tend to be much more efficient.

*We found SSAA uses Accumulation Buffer and INTEL HD Graphics no longer supports it.

2. Multisample Anti-Aliasing (MSAA)-

MSAA technique samples each pixel at the edge of the polygon multiple times. For each sample-pass a small offset is applied to all screen coordinates, which is smaller than the size of the pixel. Averaging these values the final color per pixel is calculated which helps in a smooth transition of color at the edges. Another thing to note in the multisample anti-aliasing is it runs the fragment program just once per pixel rasterized. MSAA is enabled using - glEnable(GL_MULTISAMPLE) function.

Position of sub-samples-

The position of sub samples is determined by using glGetMultisamplefv function.

```
glGetMultisamplefv( GLenum pname, GLuint index, GLfloat *val);
```

where, pname is sample parameter name. index specifies the sample whose position we have to calculate and val gives out a pointer to the position value.

The position values are between [0,1] where 0 shows the left most value of the pixel and 1 shows the right most value [0.5,0.5] will be the middle value.

- Following are the positions for 4 subsamples:

```
sample 0 y position (0.375000, 0.875000)
sample 1 y position (0.875000, 0.625000)
sample 2 y position (0.125000, 0.375000)
sample 3 y position (0.625000, 0.125000)
```

- Following are the positions for 8 subsamples:

```
sample 0 y position (0.562500, 0.687500)
sample 1 y position (0.437500, 0.312500)
sample 2 y position (0.812500, 0.437500)
sample 3 y position (0.312500, 0.812500)
sample 4 y position (0.187500, 0.187500)
sample 5 y position (0.062500, 0.562500)
sample 6 y position (0.687500, 0.062500)
sample 7 y position (0.937500, 0.937500)
```

Combination of samples to yield a final color-

The primitive rasterization produces fragments and each fragment covers some subsamples. MSAA uses the Z values, color, and coverage information to determine how to blend to the pixel's final color and work only on the edges. The number of subsamples covered by the geometry are calculated and then multiplying the percentage of coverage to the fragment color gives out the final pixel color. Hence, this gives out smooth edges.

Code for MSAA can be checked here.

3. Coverage Sampling Anti-Aliasing (CSAA)-

The CSAA mode is a technique exclusive for Nvidia GPU's. The technique improves the performance of MSAA by decoupling the coverage of triangle within a pixel from the subsamples storing the value output by the pixel shader. This work is used to determine how much each sample should contribute to the final pixel color.

*We found that the CSAA techniques runs only on Nvidia GeForce 8 Series GPU.

4. Fast Approximate Anti-Aliasing (FXAA)-

FXAA is a modern anti-aliasing technique that was developed from MSAA. FXAA is much cleverer than MSAA because it ignores polygons and line edges, and simply analyses the pixels on the screen. The biggest advantage of using FXAA is that it doesn't need that much computational power.

*We found a lot of vertex and fragment shaders resources that run FXAA but all of them use OpenGL 2.1 and GLSL 1.20 and if anyone has OSX version 10.7 or newer the oldest OpenGL it supports is OpenGL 3.2 and GLSL 1.50.