Using wesBench to Study the Rendering Performance of Graphics Processing Units

E. Wes Bethel[*]

January 8, 2010

1 Introduction

Graphics operations consist of two broad operations. The first, which we refer to here as *vertex* operations, consists of transformation, lighting, primitive assembly, and so forth. The second, which we refer to as *pixel* or *fragment* operations, consist of rasterization, texturing, scissoring, blending, and fill. Overall GPU rendering performance is a function of throughput of both these interdependent stages: if one stage is slower than the other, the faster stage will be forced to run more slowly and overall rendering performance will be adversely affected. This relationship is commutative: if the later stage has a greater workload than the earlier stage, the earlier stage will be forced to "slow down." For example, a large triangle that covers many screen pixels will incur a very small amount of work in the vertex stage while at the same time incurring a relatively large amount of work in the fragment stage. Rendering performance of a scene consisting of many large-area triangles will be limited by throughput of the fragment stage, which will have relatively more work than the vertex stage.

There are two main objectives for this document. First, we introduce a new graphics benchmark, `wesBench`, which is useful for measuring performance of both stages of the rendering pipeline under varying conditions. Second, we present its methodology for measuring performance and show results of several performance measurement studies aimed at producing better understanding of GPU rendering performance characteristics and limits under varying configurations.

First, in Section 2, we explore the "crossover" point between geometry and rasterization. Second, in Section 3, we explore additional performance characteristics, some of which are ill- or un-documented. Lastly, several appendices provide additional material concerning problems with the `gfxbench` benchmark, and details about the new `wesBench` graphics benchmark.

2 Part 1 – Measurement and Performance Analysis

2.1 Crossover Between Geometry and Rasterization/Fill Operations

Given that the graphics pipeline consists, broadly, of two stages – vertex and fragment operations – our objective here is to design an experiment that determines the relative performance of one stage to the other. As the amount of fragment operations becomes larger, the geometry rate will drop as the cost to render each triangle becomes dominated by rasterization/fill operations. If we slowly increase the area size of triangles, at some point overall triangle rate will fall off as rendering time

[*]Lawrence Berkeley National Laboratory

is dominated by rasterization/fill operations. Given that an increase of area in pixels of triangles increases the downstream fragment operation load, the objective of this experiment is to determine the point at which the workload leaving the geometry portion of the pipeline exceeds the capacity of the rasterization/fragment processing portion of the pipeline, thereby causing the vertex rate to drop.

An experimental methodology that will serve to discover the crossover point between geometry and fill operations is as follows: we'll hold constant the number of pixel operations (number of fragments) while varying the amount of geometry load and then measuring the number of (1) fragment operations per second and (2) vertex operations per second.

Approach 1: modify the `gfxbench`[1] application to iterate over triangle edge length. This approach has problems due to what seems to be conceptual design problems with the `gfxbench` application. The basic problem is that `gfxbench`'s triangle benchmark will generate triangles that lie outside the view frustum for large-area triangle sizes. The result is an artificially high triangles/second count. Those triangles that lie outside the view frustum incur a cost for transformation and lighting, but do not incur any cost for rasterization. The result, discussed later in Section 6, is that the `gfxbench` triangle benchmark generates erroneous vertex and fill rate measurements.

Approach 2: write a new benchmark, `wesBench`, that does not have these design problems, that provides a more accurate measure of geometry vs. fill rate, and that is suitable for use with the above experimental methodology. A description of this new benchmark appears later in Section 7.

2.1.1 Experiment Setup

The computational platform for all tests in this document is a dual-socket, dual-core 2.0Ghz AMD Opteron/Italy CPUs, 8GB of RAM, and an NVIDIA Quadro FX 4500[2] running SuSE 10.0 and using the 177.82 NVIDIA OpenGL driver.

In this geometry/rasterization crossover experiment, we hold the number of pixel fragments F constant and vary the geometry load and triangle area size. By holding the number of fragments constant at F, we are effectively eliminating one variable from the test (number of fragments), which is required to discover how increasing triangle area affects geometry and fill rate. Here, `wesBench` holds constant the number of fragments F at 256K. As we increase the area of triangles from 1 to 131K pixels, `wesBench` reduces the number of triangles dispatched to the graphics pipeline.

We are dispatching disjoint triangles of varying size to OpenGL using vertex arrays. While disjoint triangles won't result in peak triangle rates, we are interested in geometry rate in this test, not triangle rate. Studying the triangle rate of various triangle types is the subject of Section 2.3. In this test, per-vertex information consist of an (x, y, z) coordinate and an (R, G, B) color, all

[1]The `gfxbench` application was written by a student at Stanford University many years ago. While it does not have a formal download location on the web, it is easily findable and has been widely used as a lightweight graphics benchmark code.

[2]For this exercise, I used an NVIDIA Quadro FX 4500 graphics accelerator. Since this card is a couple of years old, a cursory search at `nvidia.com` turned up little information about the vendor's published geometry and fill rates for this card: they seem to only post specifications for current products.

A Google search turns up a couple of items:

1. A *Tom's Hardware* article (`http://www.tomshardware.com/news/nvidia-announces-quadro-fx-3450,1234.html` indicates: 181 Mtris/second and 11.3 Btexels/sec.
2. An online shopping page (`http://www.alibaba.com/product-free/12271733/Nvidia_Quadro_Fx_4500_512mb_Gddr3.html`) indicates: 181 Mtris/second and 10.8 Btexels/sec.

We'll assume that this card is capable of 181M vertex operations per second and about 11B pixel operations per second.

specified in GLfloat format. Lighting is disabled, as are basically all non-rasterization fragment operations (texturing, blending, etc.) Figure 1 shows a sample screen shot.

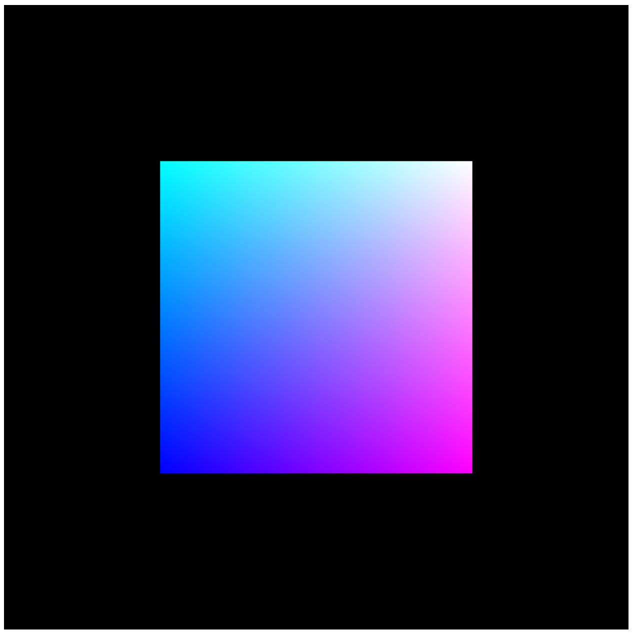


Figure 1: The `wesBench` application generates triangles that fill approximately 1/4 of the screen with triangles, and then rotates those triangles about the center of the screen over the course of the benchmark. This image shows the unlit disjoint triangles that are the basis for the geometry/rasterization crossover experiment.

2.1.2 Results

The results from this experiment, shown below in Figure 2, report the geometry rate in millions of vertices per second, and millions of fragments per second scaled by a constant value (100). This scaling normalizes the range of geometry and rasterization for the purposes of illustration: here, the important feature is the rate of change in geometry vs. rasterization rather than the absolute level of performance.

These results suggest that the crossover between geometry and rasterization occurs on this GPU as triangle area is increased from about 128 to 256 pixels – this triangle area roughly corresponds to point where the geometry performance curve begins to fall. The geometry rate for this GPU varies by about 15% with triangle area size ranging from 1..128 pixels. In Section 3.1.1, we discuss a potential reason for this variance. The geometry rate then begins to fall off sharply as triangle area is increased beyond 128 pixels.

In this test, the rasterization rate increases with triangle area over an enumerated set of triangle area sizes. The increase in rate is most dramatic between triangle area sizes of 64 to about 512 pixels, then increases more slowly. At the largest triangle area size, 131K pixels, the rasterization rate appears to be about the same "fill rate" reported by the `gfxbench` application: compare Figure 2 with 19.

One potential problem with these results and the experimental methodology is the fact that there is one additional degree of freedom that is unconstrained: the amount of geometry (number of triangles) dispatched from `wesBench` to OpenGL for each triangle area size. Given that we

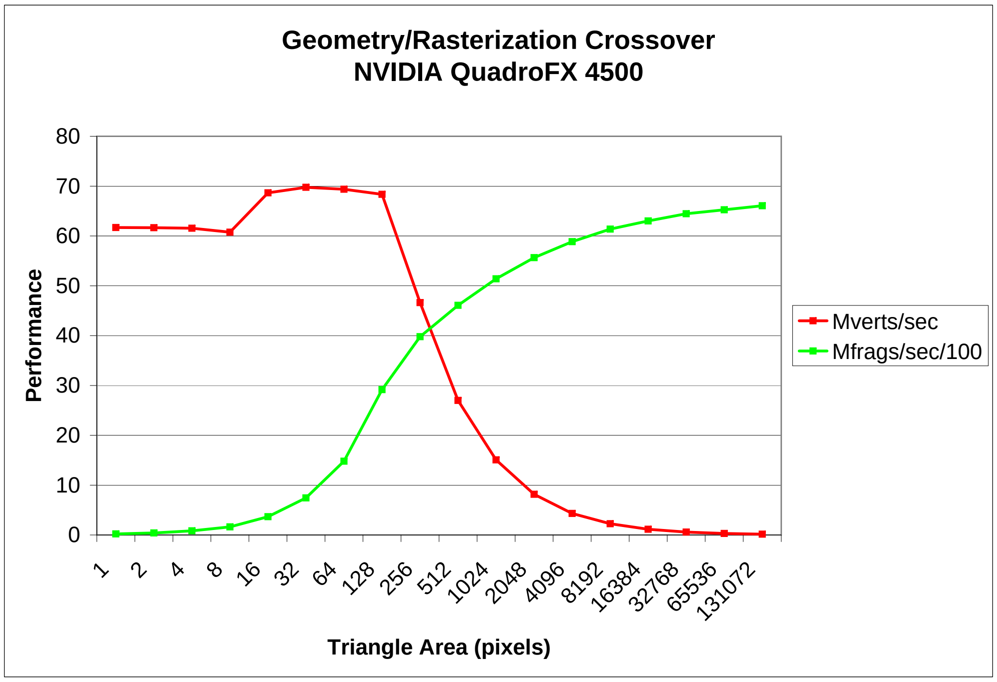


Figure 2: Comparison of geometry and rasterization performance as a function of triangle area size. The geometry rate remains high for relatively small-area triangles, then falls off dramatically after triangle area exceeds 128 pixels. This fall-off is due to the increased rasterization workload. Note that we revise our estimate of the geometry/rasterization crossover point later in Section 3.1.3, where we make more efficient use of our graphics hardware.

fix the number of fragments F to $256K$, we vary the number of triangles dispatched to produce that number of fragments as we vary the triangle area size: with large triangle area sizes, our `glDrawArrays` call has only a few triangles, while with small triangle area sizes, our `glDrawArrays` call has many triangles. It could be that overall performance might be influenced by the number of triangles per `glDrawArrays` call due to caching effects and optimal (or not) use of host-GPU bandwidth. This issue is explored later in Section 3.2.

2.2 Crossover Comparison: Lit, Textured, and Textured/Lit Triangles

This section will explore the geometry/fill crossover point for textured, lit, and textured/lit triangles. The next section will explore performance differences for different triangle types.

For these tests, we use the same test methodology and setup here for textured and lit triangles as in Section 2.1, but with the addition of texturing and lighting.

2.2.1 Lit Triangles

The lighting model in `wesBench` consists of a single positional light source co-located with the eye point. Except for specifying the light source location, we use OpenGL's default light and lighting environment parameters. The choice of a positional, rather than directional, light source is purposeful: positional lights require more per-vertex computation than directional lights.

In this test, `wesBench` dispatches disjoint triangles of varying size to OpenGL using vertex arrays. Per-vertex information consist of an (x, y, z) coordinate, an (R, G, B) color, and an (x, y, z)

normal, all specified in GLfloat format. `wesBench` computes normals such that those located in the center of the screen are parallel with the view direction, while those towards the edges become more and more perpendicular to the view direction. Figure 3 shows sample output for the lit triangles test.

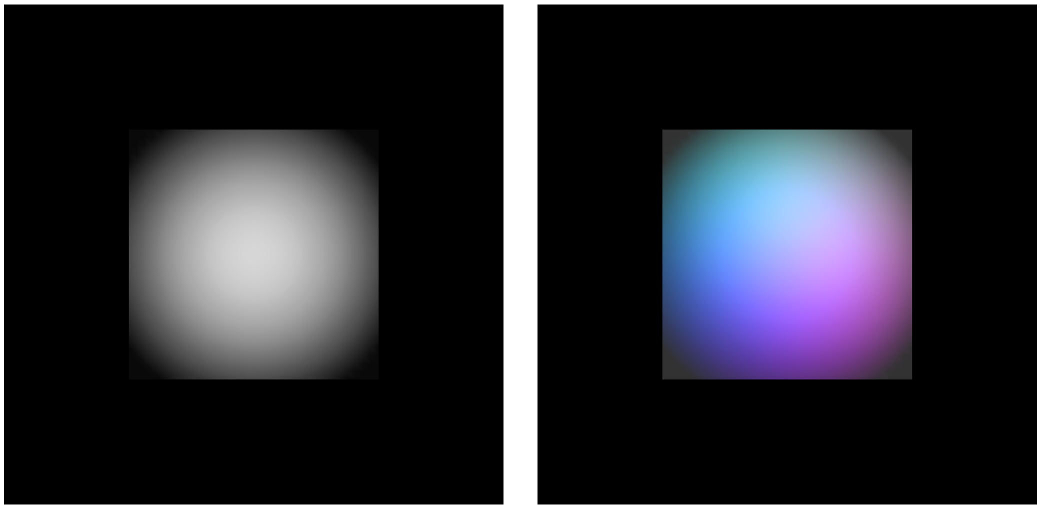

(a) Color material disabled. (b) Color material enabled.

Figure 3: These figures show the `wesBench` output when lighting is enabled. `wesBench` computes vertex normals such that those near the center are closely aligned with the view direction in eye coordinates, while those closer to the edges become more perpendicular to the view direction. OpenGL's color material attribute must be enabled for per-vertex color to be included in the shading calculation. Here, we use the per-vertex color to contribute to the diffuse portion of the shading equation.

Figure 4 shows the results comparing vertex and rasterization rates for lit and unlit triangles. When lighting is enabled, we expect the absolute vertex rate to be less than the vertex rate when lighting is disabled. This expectation stems from the fact the lighting calculation adds more work per vertex. Our results show the cost of this extra work incurs about a 30% performance drop in vertex rates up to the point where fill/rasterization begins to dominate the rendering cost. That crossover point – the location where the geometry rate begins to fall – occurs between 256 and 512 pixels of triangle area size. The difference with the unlit case, where the crossover happens between 128 and 256 pixels, is most likely due to being geometry-limited because of the added per-vertex work required to perform the lighting calculations.

There are two somewhat unexpected results in these tests. First is the minor divergence in rasterization/fill rates between the unlit and lit tests. Lighting calculations should not have any impact on rasterization/fill rates. The best explanation is that the rasterization rate is dependent in part upon the ability of the vertex engine to provide work: since the vertex rate is different between lit and unlit cases, it is reasonable to expect some variation in rasterization/fill rate when the vertex rate varies between lit and unlit tests.

The second surprise concerns the movement of a performance feature. In the unlit case, we see what appears to be a "sweet spot" for triangle areas between 16 and 256 pixels. In the lit case, that "sweet spot" has shifted a bit, showing the best performance for triangle areas between about 32 and 256 pixels. This difference is most likely due to caching effects: we are sending more data per vertex down the pipe in the lit tests than in the unlit tests, and Section 3.1.1 explores the impact

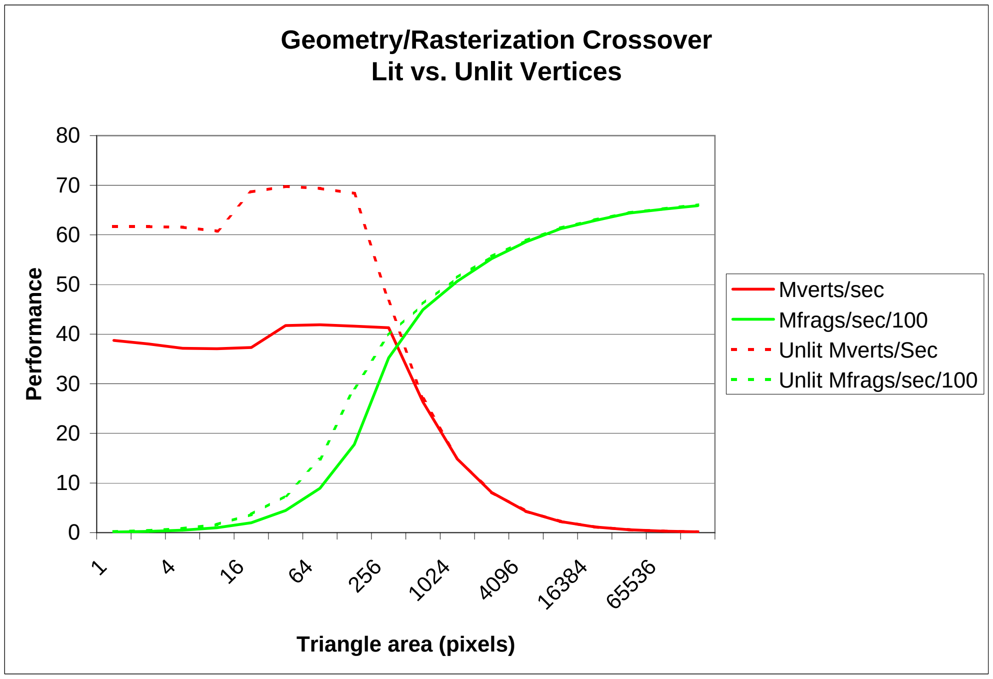


Figure 4: Test results showing geometry (red, solid line) and rasterization/fill (green, solid line) performance for lit, disjoint triangles. The per-vertex lighting calculation will cause a performance drop of about 30% when compared to geometry performance for unlit triangles (dashed red line).

on performance when varying the amount of data sent using `glDrawArrays` calls.

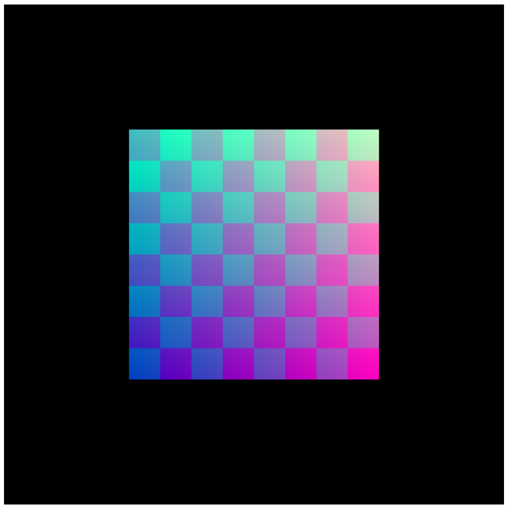
2.2.2 Textured Triangles

For the texturing tests, `wesBench` will generate a texture at a user-specified resolution[3], and will compute and dispatch per-vertex two-dimensional texture coordinates for each triangle vertex. These (s, t) texture coordinates range from [0..1] over the mesh. The texture consists of alternating red/green texels arranged in a checkerboard pattern. Figure 5 shows sample output at two different user-specified texture resolutions. We are using the GL_MODULATE texture function in all tests.

Figure 6 shows the results comparing vertex and rasterization rates for textured and untextured triangles. When texturing is enabled, we expect the absolute vertex rate to be less than that for the untextured case. The primary reason is that the texture coordinates associated with each vertex require some processing (transformation through the texture matrix stack). We see an overall performance drop of about 15% as compared to the untextured case up to the geometry/rasterization crossover point. In the case of textured geometry, the crossover point happens between area sizes of 256 and 512 pixels, which is in contrast to the unlit, untextured crossover point of 128 pixels.

We also expect the cost of the texture matrix transformation to be somewhat less than the cost of lighting: lighting requires the same vector/matrix multiplication as the texture coordinate transformation, but then also includes some additional work (a couple of vector dot products and scalar-vector operations). We see this difference reflected in the absolute difference in geometry rates shown in Figures 6 and 4.

[3]OpenGL requires textures to be an even power of two in size. Therefore, `wesBench` creates textures that are of resolution 2x2, 4x4, ..., 2Kx2K, 4Kx4K in size.


(a) 8x8 texture resolution.

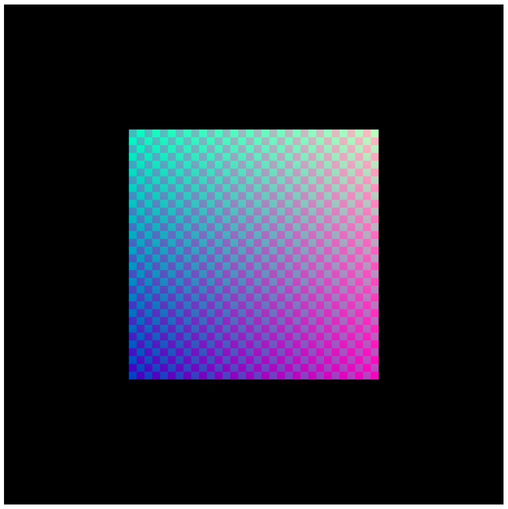

(b) 32x32 texture resolution

Figure 5: Sample `wesBench` output when texturing is enabled. The red/green checkerboard texture is combined with the per-vertex color using the GL_MODULATE texture function.

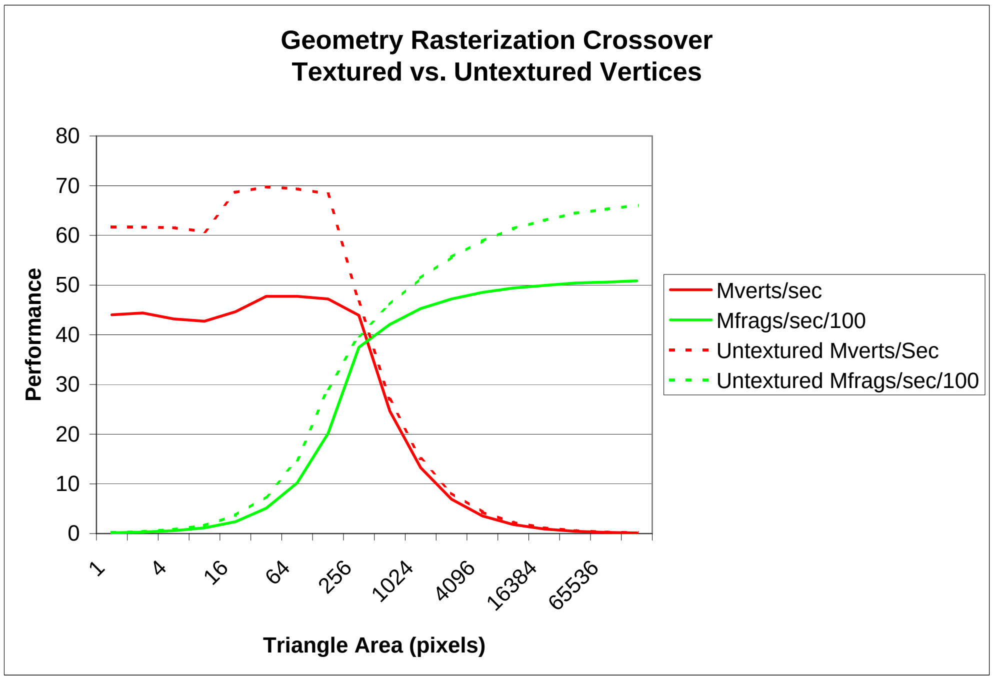


Figure 6: Results of this test are shown in solid lines: red for vertex rates and green for fragment rates. The dashed lines show vertex and fragment performance for untextured triangles. We see the geometry/rasterization crossover point is about the same for textured as untextured geometry, while the absolute performance rates are less for textured data due to the increased workload at both the geometry and rasterization stages.

Since texturing is a more expensive fragment operation when compared to untextured fragment processing, we expect the absolute fragment rate to be less for textured than for untextured geometry. The additional cost stems from the need to interpolate texture coordinates during rasterization as well as the cost of computing a fragment's texture color.

2.2.3 Textured and Lit Triangles

Whereas the lit triangle test imposes greater load on geometry processing, and the textured triangle test imposes greater load on fragment processing, doing both at the same time imposes the increased workload at both stages of the pipeline. Figure 7 shows representative output for textured and lit triangles.

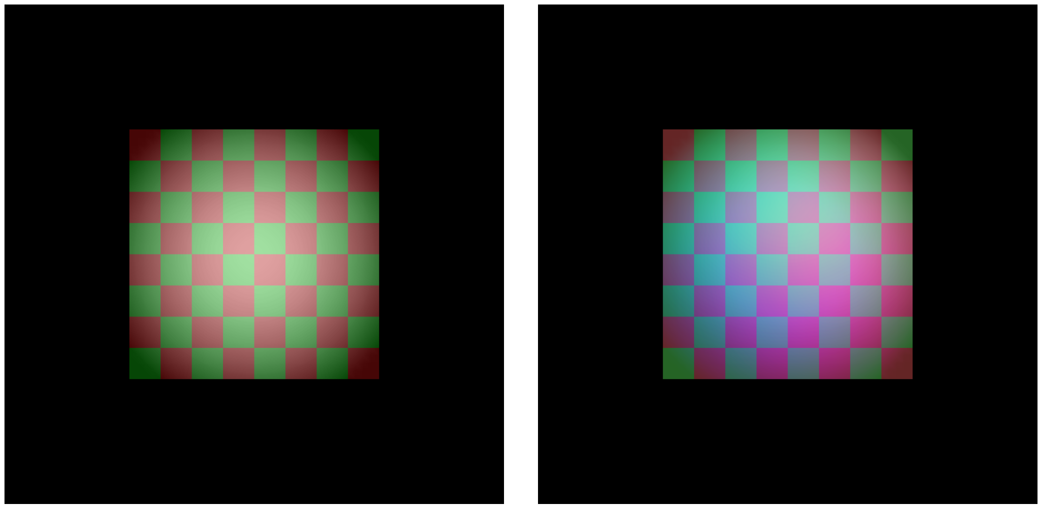

(a) Color material disabled. (b) Color material enabled.

Figure 7: Sample `wesBench` output when texturing and lighting are both enabled. When OpenGL's color material is enabled, the final fragment shade taken into account per-vertex color. The performance tests in this section were run with color material disabled.

Figure 8 shows the results comparing vertex and rasterization rates for textured, lit and untextured, unlit triangles. We see the geometry/rasterization crossover point to occur when triangle area size grows to 256 square pixels. We also see that absolute performance for the geometry stage is less in this test configuration than in previous tests. This result is due to the fact we're imposing the largest load of all tests in this discussion: transformation of geometry coordinates, use of texture coordinates, and lighting calculations for each vertex. The performance for the rasterization/fragment processing in this test configuration appears to be identical to that for the textured triangle test.

2.2.4 Discussion

For both lit and textured triangles, there is a greater amount of work per vertex than in the unlit case. This additional work results in an overall lower absolute geometry rate. As a result, the rasterization/fill portion of the pipeline is somewhat "starved" for work up until triangle area sizes reach 512 pixels and larger. In the unlit case, there is less per-vertex work for the geometry portion of the pipeline, so the rasterization/fill portion is starved for work only up to triangle area sizes of 128 pixels. When the rasterization/fill portion becomes sated with work (fill limited), geometry throughput slows.

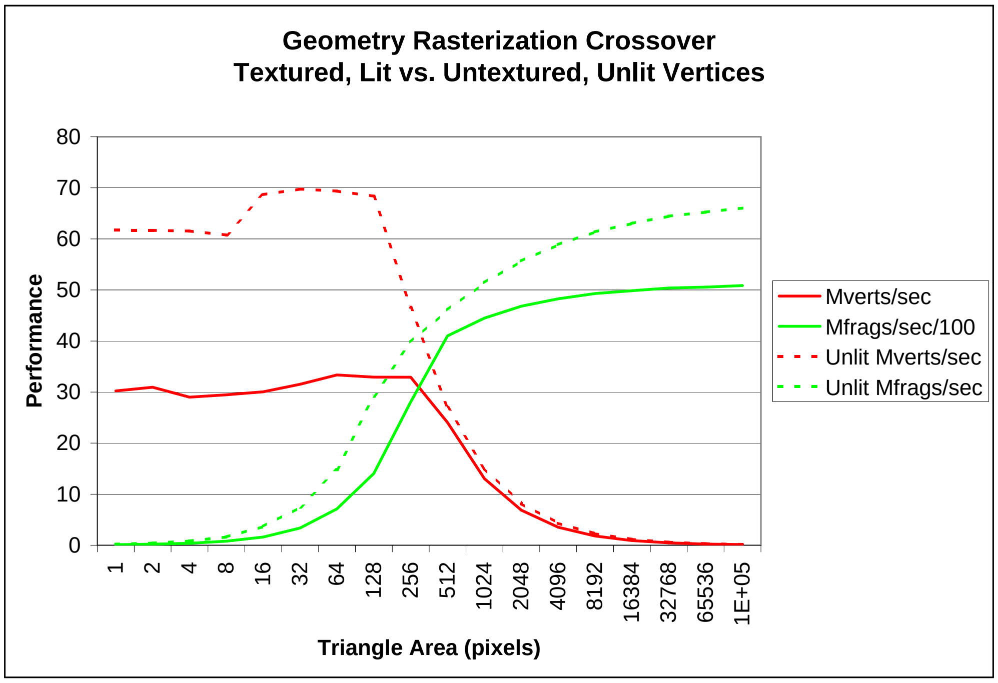


Figure 8: The cost of per-vertex lighting and texture coordinate processing for lit, textured triangles (solid red line) results in approximately a 50% performance drop when compared to unlit triangle vertex processing (solid red line). The cost of per-fragment texturing operations for lit, textured triangles (solid green line) results in a varying but significant performance drop when compared to unlit, untextured triangles (dashed green line).

2.3 Triangle Types

OpenGL offers multiple mechanisms to specify triangles for rendering: disjoint, triangle strips/fans, and indexed variants. The objective in the following sections is to evaluate the relative performance of these various alternative methods of specifying triangles. The expectation is that some types may offer performance advantages over others.

In these tests, we use the same methodology as with previous tests (see 2.1.1): we hold constant the number of fragments F, increase the area of triangles from 1 (pixel) to 131K (pixels), limit the number of triangles per area size to maintain a constant F, and additionally, vary the method for dispatching triangles to OpenGL.

We test disjoint triangles, indexed disjoint triangles, tstrips, and indexed tstrips. We did not test triangle fans because: (1) their performance will be similar to that of tstrips, and (2) tfans are not a particularly effective way of encoding triangles intended to represent an MxM lattice.

2.3.1 Disjoint Triangles vs. Indexed Disjoint Triangles

To begin, we conducted a study to compare the performance of disjoint triangles with the indexed disjoint variant.

For this problem, an MxM lattice of vertex values is decomposed into MxMx2 disjoint triangles. For the non-indexed test, a total of MxMx2x3 vertex values specify the corners of all MxMx2 triangles.

In contrast, for the indexed variant, only MxM total vertices are needed to specify the corners for all MxMx2 triangles due to the fact that these vertices are "reused" across all triangles that share a vertex. A total of MxMx2x3 *index* values are provided to OpenGL. Each value in the index array corresponds to a triangle vertex, and refers to an entry in the vertex array.

The results in Figure 9 show the performance of disjoint and indexed disjoint triangles in terms of Mverts/second and Mtris/second.

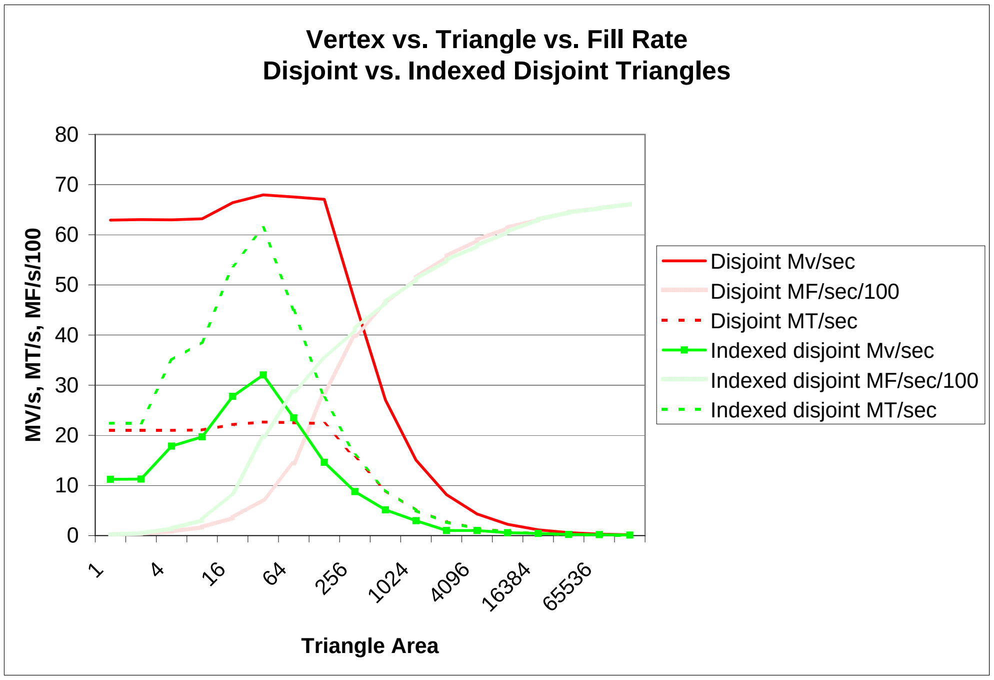


Figure 9: For this particular problem – an MxM mesh represented with triangles – the indexed disjoint triangles approach can result in an enormous geometry processing performance advantage (solid green line) and triangle rate (dashed green line) over disjoint triangles geometry processing (solid red line) and triangle rate (dashed red line). This advantage disappears after we reach the crossover point between vertex and fragment operations with triangle area size of 256 pixels and overall performance is constrained by fragment processing.

These tests confirm the expectation that the indexed disjoint triangles representation will outperform the non-indexed version in the case where there is substantial opportunity for vertex reuse. In the case of this test, each mesh vertex is shared by an average of six triangles. The performance gain, which is visible primarily for the triangle rate (dashed green and red lines in Figure 9) is most pronounced between triangle area sizes 4..64 pixels. While the absolute vertex rate seems low (solid green line), the absolute triangle rate is higher for the indexed disjoint approach due to vertex reuse.

We suspect the difference in fragment processing rate between indexed disjoint and regular disjoint triangles visible in Figure 9 due to the indexed variant's ability to keep the graphics pipeline loaded with work. In the range of triangle area sizes from 16 to 64 pixels, the indexed variant has a much higher triangle throughput, which results in more work for the lightly loaded rasterization and fragment processing portion of the pipeline.

2.3.2 Triangle Strips vs. Indexed Tstrips

To render our MxM mesh with triangle strips, we create a single large triangle strip as follows: for each of the M-1 rows of our MxM mesh, a total of 2M vertices specifies 2M-2 triangles, and we add one "degenerate vertex" at the end of one row and the beginning of the next row. The result is that each of the M-1 rows has on average four "degenerate triangles" – these have two triangle vertices co-located. The expectation is that these degenerate triangles will incur a negligible amount of vertex processing cost, but will incur zero fragment processing cost (when using GL_FILL for the polygon mode) since the triangles have zero area.

The largest benefit of this approach – using degenerate triangles to stitch together tstrips from each of the mesh rows – is that we can dispatch the entire mesh off to OpenGL with a single call. Later in Section 3.1.1 where we study the potential performance impact of varying the number of vertices per call, no special processing is required for triangle strips.

To represent an MxM mesh using triangle strips requires approximately $2M^2 + 2M$ vertices (compare to $6M^2$ vertices for disjoint triangles): each of the M-1 mesh rows requires about 2M vertices, and there are about 2M's worth of degenerate vertices. For the indexed disjoint tstrips, there are M^2 vertices and about $2M^2 + 2M$ index values – about half as many vertex values are processed when using indexed tstrips as compared to tstrips.

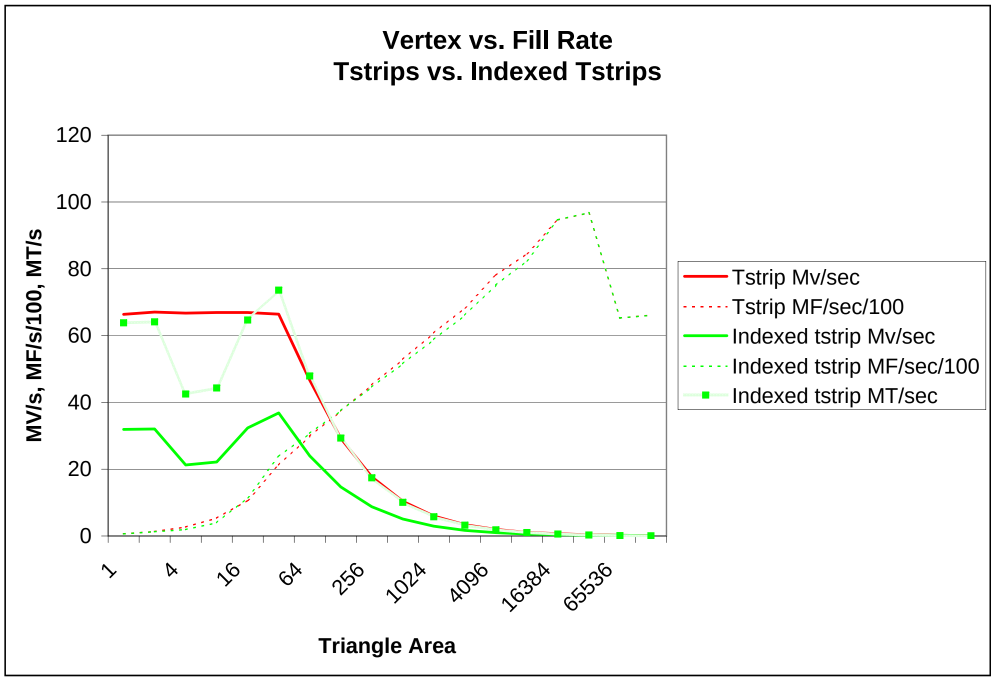


Figure 10: Comparison of vertex rates for triangle strips (solid red line) and indexed triangle strips (solid green line) along with rasterization/fill rate for triangle strips (dashed red line) and indexed triangle strips (dashed green line). For the regular tstrips, absolute triangle rate is nearly identical to the vertex rate and is not shown here. In contrast, the triangle rate for indexed tstrips is shown as the fuzzy green line with markers. Except for the indexed tstrip performance "sour spot," which corresponds to triangle area sizes of four and eight pixels, there is not much triangle rate performance difference between tstrips and indexed tstrips.

However, when we look at the test results comparing performance of tstrips and indexed tstrips (Figure 10), there is not as much of a performance difference as one might expect. Except for the indexed tstrip performance "sour spot" corresponding to triangle area sizes of 4.0 and 8.0 pixels[4], there really doesn't seem to be much difference in performance between tstrips and indexed tstrips.

Figure 10 reports vertex rate and rasterization/fragment processing rate over a number of triangle area sizes for unlit, untextured triangle strips and indexed triangle strips. It turns out that triangle rate for regular tstrips is more or less identical to the vertex rate, so there is no separate curve in the Figure 10 plot. The triangle rate for indexed tstrips is quite a bit larger than the vertex rate due to vertex reuse. We simply don't see the same dramatic performance gain for indexed tstrips that we saw earlier with indexed disjoint triangles.

An interesting feature in Figure 10 is the outstanding fill rate performance when using larger-area triangles. The `gfxbench` application reports a fill rate of about 6500Mpix/sec on this hardware; we are seeing in excess of that rate for triangle areas greater than 2048 pixels^2, with a rate that is about 90% of the vendor's numbers for triangle area sizes of 16384 and 32768 pixels.

As the triangle area size increases, the number of triangles per strip decreases. We see fill rate continue to increase up to triangle area of 32768 pixels. At that level, the triangle strip has 14 vertices (the indexed variant has nine vertices and 14 indices). At area sizes of 65536 and 131072 pixels, after which the fill rate falls off dramatically, the tstrip has only four vertices. We speculate that there is a relationship between the size of the batch of triangles in a strip and the ability for the rasterization/fill engine to remain filled with work. There may be some one-time triangle setup costs that can be amortized across many triangles in a strip, and this amortization is resulting in increased rasterization/fill efficiency.

2.3.3 Overall Comparison

In Figure 11, we have the triangle rate performance data for all triangle types in a single chart. From this chart, we can draw several observations. First, triangle performance with tstrips is roughly three times better than disjoint triangles in our testing up until the point where the workload becomes constrained by rasterization/fill operations. This result is not unexpected.

Second, somewhat unexpected is the unusual variation in triangle rate for the indexed variants. The indexed tstrip variant exhibits an unusual "sour spot" that is not explained by any of the tests in this entire document. Our best guess is a bug in the driver code that moves array element data from the host. The other unusual variance is the variance in indexed disjoint triangle rates prior to the geometry/rasterization crossover. For smaller-area triangles, more work is required in memory lookups than for larger-area triangles. There may be an internal cache buffer associated with these dereferenced memory locations that is overflowed, and combining this cache limit with the potential bug resulting in the indexed tstrip sour spot, we see unusual variance in triangle rates for these indexed variants.

Refactoring the triangle rate a bit, Figure 12 shows the relative rather than absolute performance in triangles/second of all triangle types. In this Figure, the y-axis represent percent of maximum of all triangle types. We see that tstrips dominates performance except for a couple of triangle area sizes, where indexed tstrips excels. The performance of disjoint triangles ranges from a low of about 30% of tstrips for small area triangles up to identical at the largest triangle sizes. The relative performance difference between the different types is difficult to see due to scale on the absolute Mtris/sec chart (Figure 11).

Even for the larger triangle area sizes, we see a significant difference in performance between the disjoint and tstrip representations. Even though these differences are relatively small in absolute terms, they are more visible when presented as a percent-of-maximum. One conclusion here is that

[4]This sour spot was suspicious – we reran this test several times and can confirm the sour spot makes an appearance across several different test runs.

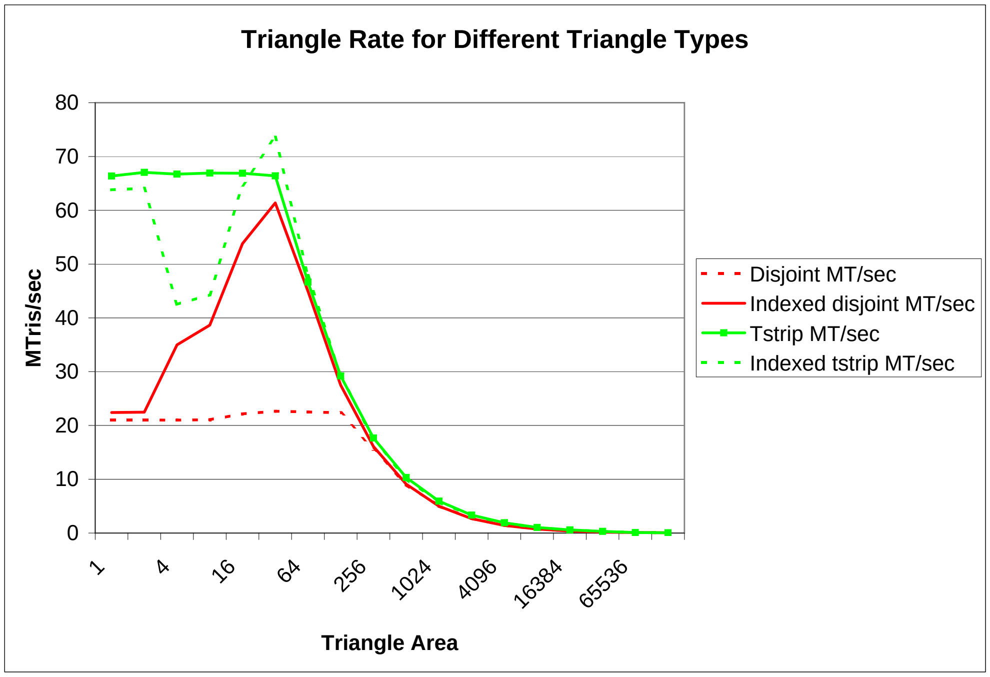


Figure 11: Test results showing absolute Mtris/second of four different ways of dispatching triangles to OpenGL. Up to the geometry/rasterization crossover point, which occurs at about 128 pixels for tstrips and 256 pixels for disjoint triangles, we see substantial variance in triangle rate when using alternative methods for specifying triangles that represent an MxM mesh to OpenGL. After the geometry/rasterization crossover point, triangle rendering rate is dominated by rasterization/fragment operations, and the mechanism for specifying triangles to OpenGL is no longer a factor on absolute triangle rate performance.

there is likely triangle setup work that is amortized across the multiple triangles of a tstrip, and that that amortization results in a cost savings and performance gain even at the relatively low levels of absolute triangle rate performance associated with fill-dominated, large-area triangles.

3 Part 2 – Additional Performance Factors and Undocumented Features/Limits

3.1 More Performance Characterization

3.1.1 Impact of Bucket Size on Performance

One potential tunable application attribute is the number of vertices that are “moved” from the client into the server for each `glDrawArrays` call. Based on others’ experience, we believe and expect that sending too few vertices per call will not make effective use of bandwidth to the GPU. This inefficiency will be realized as relatively low performance. However, we wonder if there is a “sweet spot” that balances sending enough vertices to make efficient use of bandwidth against potential buffer/cache overflow at various locations from the host down through the graphics pipeline.

For this experiment, we measured vertex rates when dispatching varying numbers of vertices using `glDrawArrays`. The results of this test, shown in Figure 13, confirm the expectation that sending too few vertices per call results in relatively poor performance. Starting with relatively few

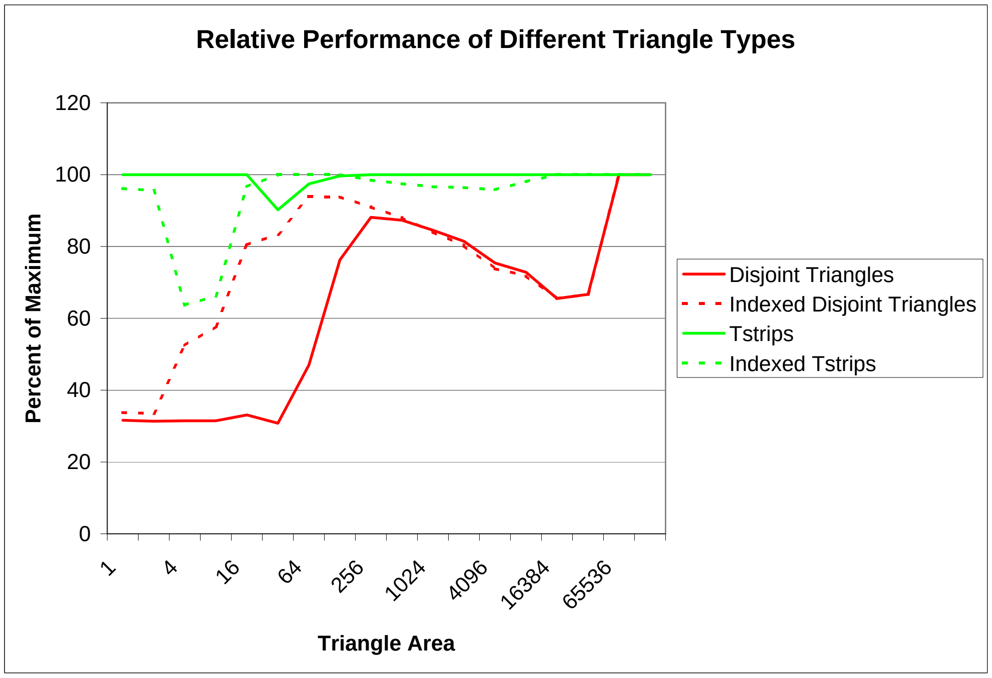


Figure 12: Here we see triangle rate expressed as the percent-of-maximum of all different triangle types. At a glance, we see that tstrips dominates performance (at 100%) for all but two triangle area sizes.

vertices per call/bucket, we see a rapid performance increase as we increase the number of vertices per bucket.

We ran three different test batteries. The first is with unlit disjoint triangles: the amount of data per vertex is (x, y, z) position and (r, g, b) color in GLfloat format, for a total of 24 bytes/per vertex. The second test adds texture coordinates per vertex, or an additional 8 bytes per vertex. The third test adds per-vertex normals, each of which is (x, y, z) GLfloats, or an additional 12 bytes per vertex. These tests were run with a very small triangle area – 0.03125 pixels. Therefore, the number of fragments is basically the same as the number of triangles: each triangle will generate one fragment given that the triangle has sub-pixel area. We ran these tests using triangle strips so as to capitalize upon their superior performance characteristics.

For the unlit triangles, we see performance rise rapidly with increasing numbers of vertices/bucket, then peak and level off at 1K vertices per bucket. This threshold corresponds to a total of 1K*24bytes/vertex or a total 24KB memory footprint for the geometry payload. After 32K vertices per bucket (768KB memory footprint), performance drops by about 10% then levels off 128K vertices/bucket (3MB memory footprint) and remains flat out to 8M vertices/bucket.

For the textured triangles, we also see performance rise rapidly with increasing numbers of vertices/bucket. Performance peaks and levels off at 1K vertices (1K*32bytes/vertex, or 32KB of payload). Performance remains mostly flat out to 32K vertices/bucket (1MB memory footprint), at which point it drops by about 10% and remains flat out to 8M vertices/bucket.

For the lit, textured triangles, we again see performance rise rapidly with increasing numbers of vertices/bucket. Performance peaks and levels off at 512 vertices/bucket (512*44bytes/vertex is about 22KB of payload). Thereafter, performance remains flat out to 16K vertices/bucket (704KB memory footprint), then drops about 10% and remains flat out to 8M vertices/bucket.

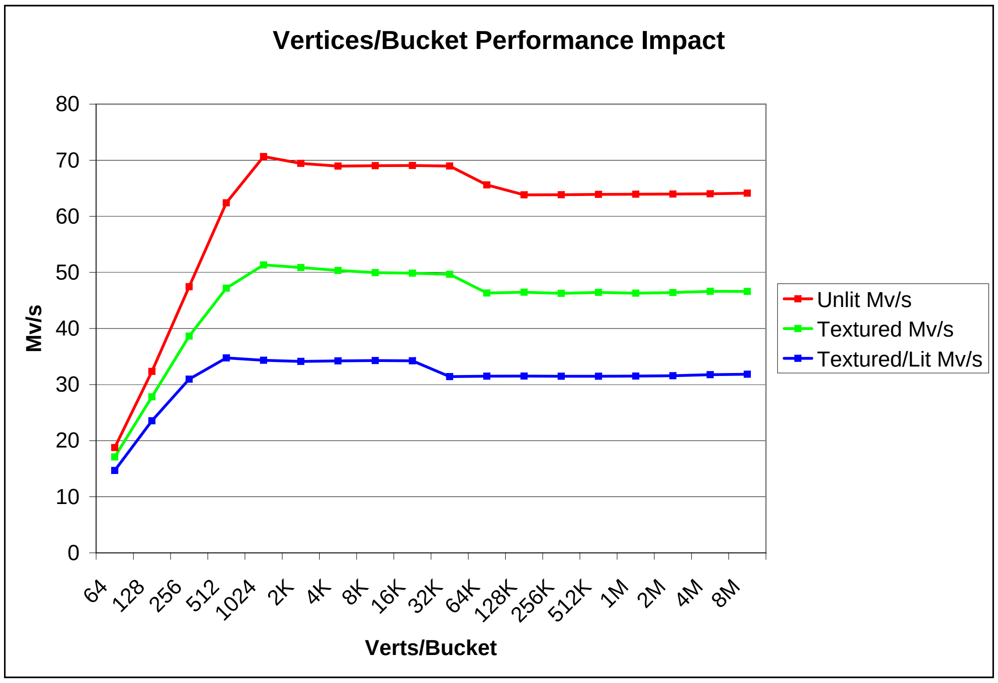


Figure 13: The GPU's vertex throughput rate is strongly correlated with the number of vertices dispatched with each `glDrawArrays` call. Performance is worst when dispatching relatively few vertices per call.

The relative performance difference between the curves is not a concern: we already established that our test platform processes twice as many unlit, untextured vertices per unit time than it can process lit, textured vertices.

From this information, we conclude there are two different thresholds at play here. The first is that we need to issue at least 22KB worth of payload per bucket to reach "peak" levels of performance. This performance threshold is likely a function of host bus bandwidth to the GPU. The second threshold – marked by about a 10% drop in vertex performance – seems to happen when the amount of data per bucket exceeds between 700KB and 1MB (more testing would be needed to refine this estimate).

Therefore, the "sweet spot" for performance on this platform is to issue between about 32KB and 700KB worth of payload per bucket. The contents of that payload can be any combination of vertex position, color, or texture coordinates.

Given this insight, we can now make observations about features seen in earlier tests. Going back to Figure 2 looking at the crossover between geometry and fill performance, we see an interesting feature on the Mverts/sec curve: for small triangles, performance is more or less steady at about 60Mv/sec for triangle areas 1 to 8 pixels. Then, there is a performance jump to about 70Mv/sec for triangle areas 16 to 128 pixels. At triangle areas of 256 and higher, performance drops off dramatically.

We have established that the performance dropoff at larger triangle area sizes is due to increased rasterization workload. The slightly lower levels of performance for smaller triangle area sizes correspond exactly to the lower levels of performance on the vertices/bucket chart in Figure 13.

We can correlate this observation more precisely with the results in Figure 13. For the 16 pixels triangles, `wesBench` sends 48,600 vertices. This amount of payload represents about a 1.1MB memory footprint. Increasing the triangle area results in fewer triangles per call, and a smaller memory footprint for the payload. Decreasing the triangle size smaller than 16 pixels results in payload having more than a 1.1MB memory footprint. Therefore, we expect performance for small area triangles, where the size of geometry payload well exceeds 1MB, to be about 10% less than larger area triangles, where the size of the geometry payload is about 1MB or less. This expectation is confirmed in several of the previous figures in this discussion (e.g., Figures 2, 4, 6, 8, 9, and 11.)

3.1.2 Vertex Arrays and Display Lists

In earlier work[5], we observed an unusual performance characteristic on several different modern GPUs: dispatching data to the GPU using vertex arrays exceeded performance as compared to either display lists containing `glVertex/glColor` calls or display lists containing vertex arrays. The difference was not small: 6.1Mtris/sec for display lists with `glVertex/glColor` calls, 5.8Mtris/sec for display lists with vertex arrays, and 113.8Mtris/sec for vertex arrays. These results are for a dataset consisting of about 2M disjoint triangles with per-vertex normal data, a single scene light, and about half the fragments being obscured in depth by other fragments.

We added a `-retained` flag to `wesBench` to explore enclosing our vertex array calls inside display lists and testing performance. When encapsulating vertex array calls inside display list, we see an across the board performance drop of about 50% as compared to just making vertex array calls.

The question is "why is this the case?" Some preliminary investigation turns up the following clues:

- From the FAQ OpenGL website[6]: "Will putting vertex arrays in a display list make them run faster?" Answer: It depends on the implementation. In most implementations, it might decrease performance because of the increased memory use. However, some implementations may cache display lists on the graphics hardware, so the benefits of this caching could easily offset the extra memory usage. Comment: on all OpenGL implementations we have tested, we have determined that putting vertex arrays inside display lists is a sure-fire way to degrade performance. Therefore, someone at the OpenGL website might need to update the FAQ.

- While display lists are immutable once set, vertex array data may change from call to call. OpenGL must take extra steps to determine if any of the array data has changed to decide if it needs to copy it to server memory. There was a clue on a discussion board[7] that suggests this check may incur some overhead. There was no further information about the amount of overhead, nor details about what "checking for changes" entails.

- In the quest for performance, the next thing to investigate is Vertex Buffer Objects. A nice discussion is located on `spec.org`[8]. In particular, VBO's offer flags that indicate whether or not buffer data will change. If data is marked as static, it can be cached in GPU memory. Presumably, this approach will result in optimum performance, and is really the desired target for encapsulating vertex arrays inside display lists. VBO's are likely a viable approach now for they were included in OpenGL starting in version 1.5.

In short, our investigation was not successful in locating conclusive facts nor discovering performance features through testing that reveal why performance is relatively poor when encapsulating vertex arrays inside display lists.

[5]See the *svPerfGL* benchmark, `http://vis.lbl.gov/Research/svPerfGL`

[6]http://www.opengl.org/resources/faq/technical/displaylist.htm

[7]`http://www.allegro.cc/forums/thread/300150`

[8]`http://www.spec.org/gwpg/gpc.static/vbo_whitepaper.html`

3.1.3 Vertex Buffer Objects and Vertex Arrays

In the quest for maximum performance, we explored the use of vertex buffer objects (VBOs) as an alternative to vertex arrays (VAs). In brief, the VBO allows VA data to be stored in high-performance graphics memory on the server side (e.g., GPU memory). VBOs support a number of usage flags that help the OpenGL implementation to optimize VA data placement and usage. One form, GL_STATIC_DRAW_ARB, says the array data will never be modified: the data can be cached and never checked for updates. Another form allows the application to modify the VA data.

We have two questions in mind: (1) does use of VBOs result in higher performance than use of VAs? (2) how is performance impacted by encapsulating VBOs inside a display list?

To answer the first question, we created a modified version of the benchmark named `wesBench-vbo`, which makes use of VBOs. We ran a battery of tests to collect performance data while varying triangle area. Figures 14 and 15 show the results of these tests for disjoint triangles and tstrips, respectively.

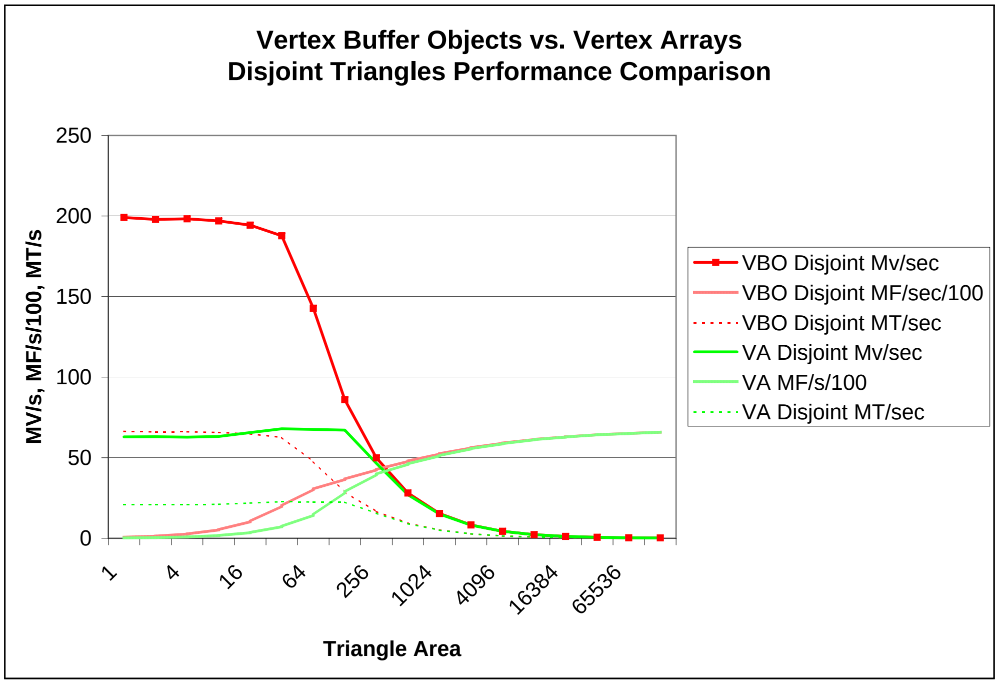


Figure 14: Comparison of geometry, triangle and fill rate when using VBOs as compared to vertex arrays for disjoint triangles. Using VBOs, we are able to reach the maximum possible geometry rate on our test system, and are able to achieve three times the triangle rate of vertex arrays up to triangle area sizes of about 32 pixels, which seems to be the geometry/rasterization crossover point for this GPU.

Figure 14 shows some interesting results. First, we observe about a three-fold performance increase in terms of both geometry operations and absolute triangles/second for VBOs over VAs for triangle sizes in the range 1..32 pixels. The reason for this massive performance increase is because the VA data is being cached in high performance memory on the GPU rather than, presumably, being copied over the PCIe bus on each frame.

The second interesting result is that geometry rate falls off rapidly between 32 and 64 pixel^2 triangles when using VBOs as compared to 128 and 256 pixel^2 triangles when using VAs. Based upon this observation, we can now correct our claim in Section 2.1 about the location of the geometry/rasterization crossover. Those earlier results were biased: we weren't utilizing our graphics hardware's geometry engine to its fullest potential, so the crossover rate was "artificially" high in terms of triangle area since the geometry engine could not sufficiently saturate the rasterization/fill engine with work, so the geometry rate began to fall off rapidly after going larger than 128 pixels in triangle area. Here, we see the falloff happen at a much earlier location, and now revise our crossover point estimate: this hardware seems to be optimized for 32 pixel^2 triangles: at larger area sized, the absolute triangle rate begins to fall.

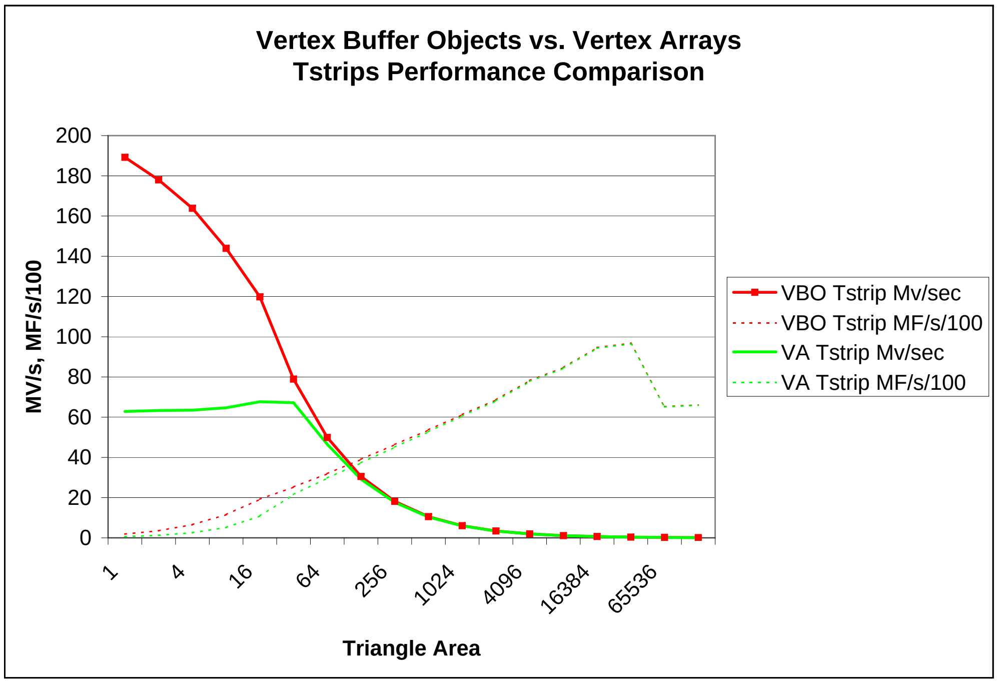


Figure 15: Comparison of geometry and fill rate when using VBOs as compared to vertex arrays for tstrips. For tstrips, the triangle rate is nearly identical to vertex rate and is not included in the chart. Up to 32 pixels area sizes, the VBO tstrips outperform VA tstrips by a substantial amount (at least 10%, at most about 300%).

Figure 15 shows the results of the VBO vs. VA tests when using tstrips. As with the disjoint triangles, VBO performance far exceeds VA performance for small-sized triangles. Starting at 64 pixels triangles, VA and VBO tstrip performance is nearly identical, which suggests that rasterization/fill costs begin to dominate at that triangle size. This observation seems to confirm our earlier claim that the geometry/rasterization crossover occurs at 32 pixel^2 triangles.

In answer to the second question, "how is performance impacted by encapsulating VBOs inside a display list," we ran a few point-sample tests. In all of those tests, encapsulating VBOs inside a display list resulted in a 30% performance decrease as compared to straight VBOs.

3.2 Discovering Undocumented Features/Limits

3.2.1 Texture Image Size Impact on Performance

In this test, we want to determine if the size of a texture image can have an impact on frame rate. Also, we want to determine if texture image resolution (i.e., number of texels) or the memory footprint of a texture image (e.g., internal storage format) has more or less of an impact on performance.

For this test, we ran `wesBench` using all triangle types (disjoint, tstrips, indexed disjoint, indexed tstrips), a triangle area size of 64 pixels, and varied the texture image size and internal storage format. We varied the texture image resolution from 8x8, 16x16, ..., 4096x4096 texels. Table 1 shows the memory footprint for the texture image at various resolutions and storage formats.

Texture Size	Texels	GL_RGBA8 size (KB)	GL_R3_G3_B2 size (KB)
8x8	64	1/4	1/16
16x16	256	1	1/4
32x32	1024	4	1
64x64	4096	16	4
128x128	16K	64	16
256x256	64K	256	64
512x512	256K	1024	256
1024x1024	1024K	4096	1024
2048x2048	4096K	16384	4096
4096x4096	16384K	65536	16384

Table 1: The memory footprint for the different resolution textures varies from 1/16KB up to 64MB across the range of texture sizes and storage formats.

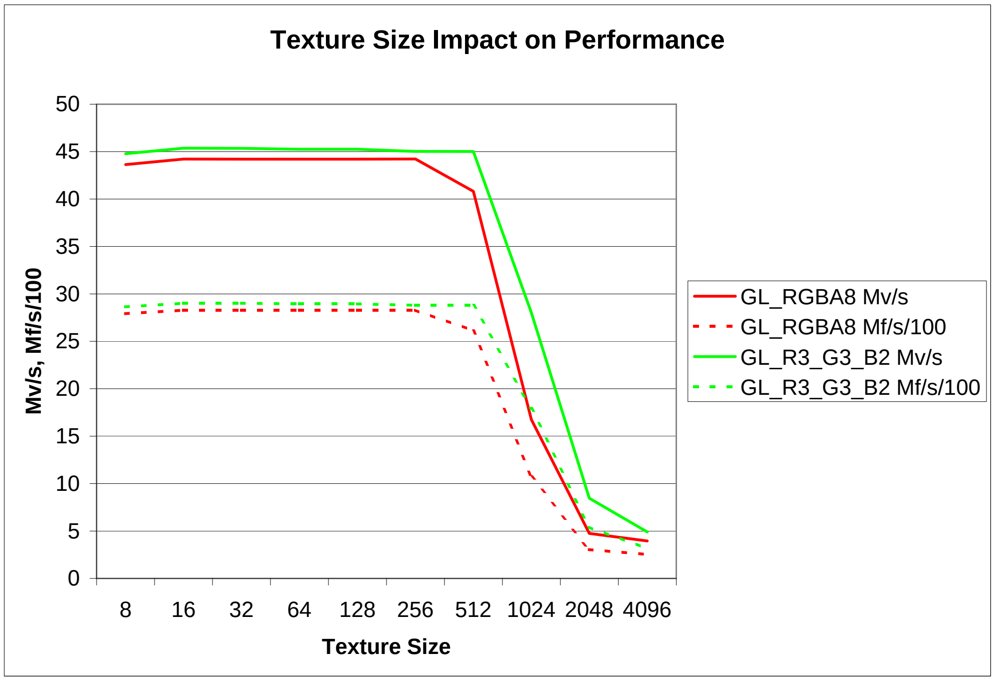


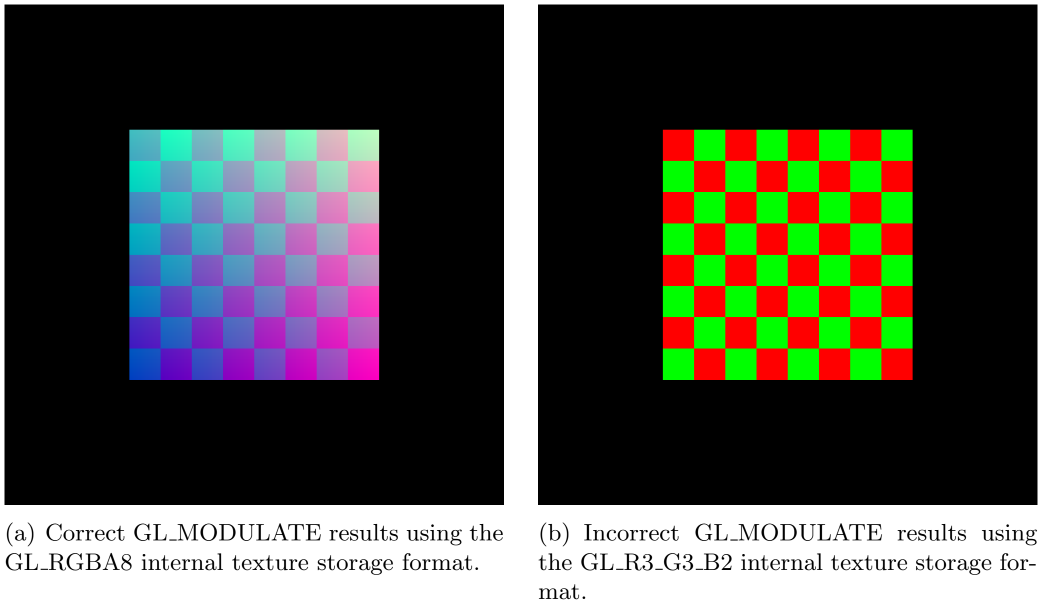
Figure 16: Varying the texture image size/resolution and internal storage format can have an impact on both geometry and pixel rate. In this figure, the red curves show performance when using the `GL_RGBA8` and `GL_R3_G3_B2` internal storage formats (4- and 1-byte per texel, respectively). The solid lines show geometry performance, and the dashed lines show pixel rate performance.

Figure 16 shows the results of this battery of tests. We see that geometry and fill rates seem to be more or less unaffected by texture image resolution or storage format up to a limit: we see a performance degradation starting at texture resolutions beyond 256x256 texels for the `GL_RGBA8` format and beyond the 512x512 texels for the `GL_R3_G3_B2` format. Since these two transition points represent a memory footprint of 256KB, we conclude that optimal texturing performance on this GPU occurs when textures have an internal memory footprint of less than or equal to 256KB. Beyond that size, texturing performance degrades as less and less of the texture fits into the texture cache. We conclude that the texture cache size on this GPU is 256KB.

3.2.2 NVIDIA OpenGL Driver Bugs – Incorrect Rendering Using GL_R3_G3_B2 Internal Texel Storage Format

In the early days of the NVIDIA Linux OpenGL drivers, it seemed we were reporting several bugs to NVIDIA with each new driver release. We have not reported any bugs to them for many years, a fact that reflects the maturation of the driver technology. To our surprise, we ran across a rare rendering bug while doing the tests for this project.

The bug, shown in Figure 17, is that textures having an internal storage format of `GL_R3_G3_B2` and displayed using `GL_MODULATE` are incorrectly rendered using what appears to be `GL_REPLACE` operation.



(a) Correct GL_MODULATE results using the GL_RGBA8 internal texture storage format.

(b) Incorrect GL_MODULATE results using the GL_R3_G3_B2 internal texture storage format.

Figure 17: The left image shows the correct, expected results when using the `GL_RGBA` internal storage format: `GL_MODULATE` multiplies the incoming fragment color with the texture color. On the left, the fragments from the base mesh range from blue (lower left) to magenta (lower right) to white (upper right) to cyan (upper left), and the texture is correctly modulated with those fragment colors. On the right, there is no modulation – it looks like the incoming fragment's color is simply replaced with the texture color, which is how `GL_REPLACE` operates.

4 Lessons Learned/Conclusions

The `gfxbench` application has a number of conceptual design problems that make it unsuitable for use in studying the performance of GPUs in some problem configurations (see Section 6), and as a result, we created a new benchmark `wesBench`. This new benchmark does the right thing with respect to triangle rate, vertex rate, and fill rate calculations, and is also more easily extended to accommodate the various other tests included in this discussion.

The initial performance runs aimed at discovering the geometry/rasterization crossover point used vertex arrays (as does `gfxbench`) for dispatching triangle data to OpenGL. It wasn't until quite late in testing, when frustration with poor performance of vertex arrays inside display lists led to the exploration of VBOs as a high performance option, that the validity of these early tests was questioned. It wasn't until seeing the geometry pipe run at 100% of capacity, possible only with the VBOs, that the 32 pixel^2 triangle emerged as the most likely crossover point for this graphics hardware.

Unfortunately, extensive testing in this exercise did not provide any definitive clues to answer the question "why does encapsulating VBOs or VAs inside a display list produce a dramatic performance drop?"

The memory footprint size of a texture can impact performance. Test results in Section 3.2.1 suggest a 256KB texture cache: performance drops significantly when using textures having a memory footprint larger than 256KB.

The amount of data sent per call using the VA calls can have a noticeable impact on performance. Testing in Section 3.1.1 suggests low- and high-water marks for bucket sizes. A least 22KB of data per VA call is needed to make effective use of bandwidth; less than that and performance suffers. Sending more than about 1MB per call will result in about a 10% performance drop as compared to the 22KB-1MB bucket size. We ran a similar set of tests using VBOs (results not shown); those test results exhibit a similar performance profile to the ones run using VAs, but the VBO tests have significantly higher levels of performance compared to the VAs.

We discovered what appear to be two bugs in the NVIDIA driver. The first is a rare rendering bug: GL_MODULATE does not appear to work when using the GL_R3_G3_B2 texture (see Section 3.2.2). The second is a "sour spot" in performance associated with indexed triangle strips (Section 2.3.2). It turns out this "sour spot" happens only with indexed triangle strips using VAs: when using VBOs, there is no performance "sour spot." This difference suggests a problem with managing and moving element array data between the client and server.

5 Acknowledgment

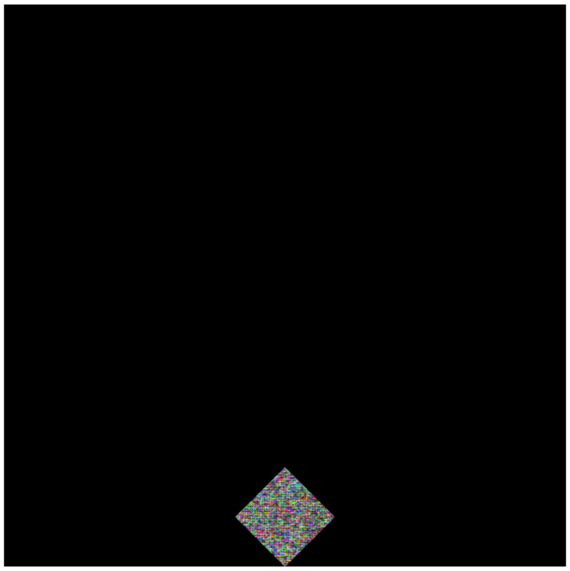
This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET).

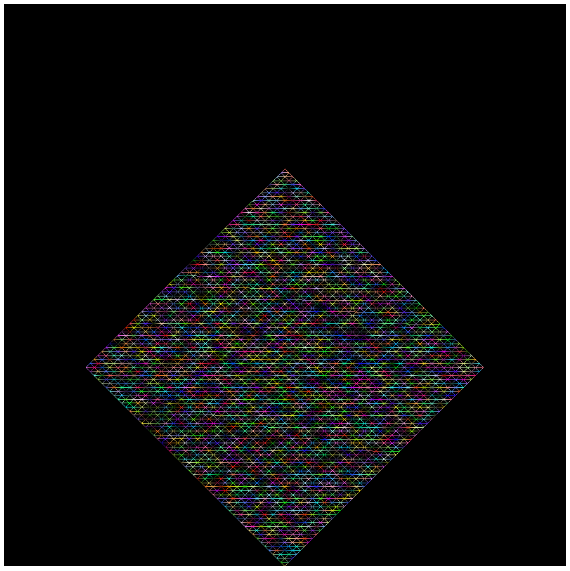
6 Appendix A – Problems with gfxbench

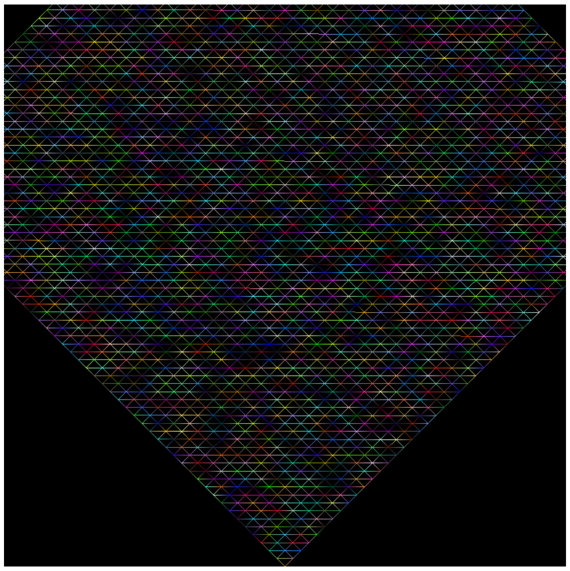
For the triangle benchmark, `gfxbench` creates a 50x50 mesh where triangle edge length is a user-specified parameter. Then, this triangle mesh is rotated over some number frames. At the end of a period of testing, the total number of triangles rendered is divided by time to produce an estimate of triangles per second.

The problem is that this computed rate is artificially high due to a fundamental design flaw.

This flaw is as follows: at larger triangle edge sizes, not many of the triangles are actually visible. Figure 18 illustrates this problem. As a result, these off-screen triangles will be drawn really fast since they incur zero fragment operations, and the resulting triangle rate is artificially high.


(a) Edge size=2


(b) Edge size=8


(c) Edge size=16

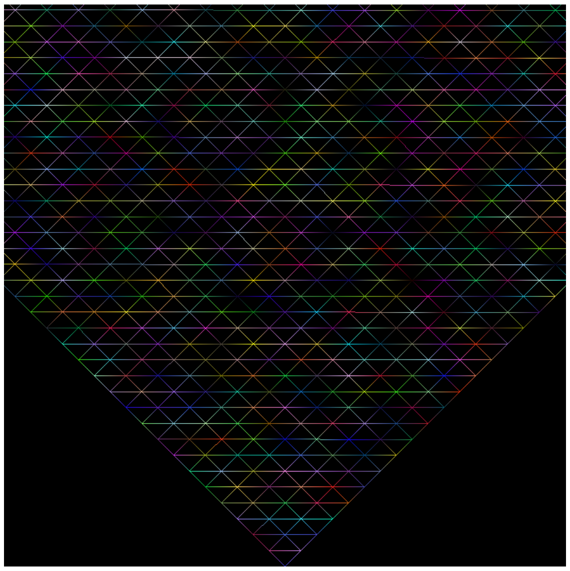

(d) Edge size=32

Figure 18: These screen grabs from the gfxbench triangle benchmark, modified to use polygon outline mode to clearly show triangle boundaries, show that as the triangle edge size is increased, that more and more triangles fall outside the view frustum. Since the culled triangles incur no rasterization or pixel processing load, the computed fill rate is not correct – it is artificially high.

To illustrate the severity of this problem, Figure 19 shows the fill rate computed by `gfxbench` over a number of triangle edge sizes. This fill rate starts out at reasonable levels for small edge sizes. At an edge value of about 17 or so, the reported fill rate exceeds the fill rate `gfxbench` measured with its own fill rate benchmark. Then, at about an edge value of about 24 or so, the measured fill rate exceeds the vendor's marketing numbers. The measured fill rate continues to grow without bound as edge size increases.

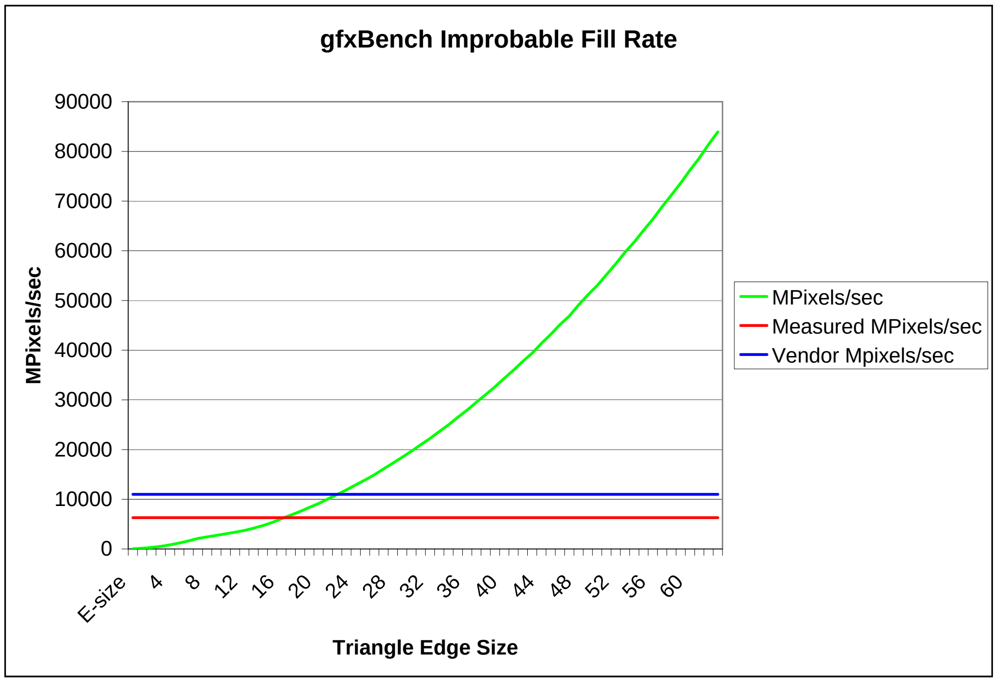


Figure 19: Due to an internal design flaw, `gfxbench` measured fill rate from its triangle benchmark generates impossibly high fill rate numbers, numbers that far exceed the vendor's own marketing claims.

7 Appendix B – wesBench Graphics Benchmark

The `wesBench` bench operates by constructing an MxM mesh of points that are positioned so they fill 1/4 of the screen resolution. This fraction is chosen so that all fragments remain visible over the course of the triangle benchmark, which consists of rotating the mesh about the center of the screen for user-specified time duration.

The spacing of vertices is computed to produce triangles of a user-specified area. If the user specifies a triangle area A, then the mesh points are spaced at $sqrt(2A)$ intervals. Some area sizes result in less than 1/4 of the screen being filled with triangles. In all instances, all triangles are visible, so the resulting statistics (Mverts/sec, Mtris/sec, Mfrags/sec) are computed using the actual number of triangles/vertices that are displayed. For this reason, the statistics produced by `qwesBench` are valid, whereas the `gfxbench` tool produces bogus statistics for some triangle area sizes.

Inside the triangle benchmark routine, we first compute the MxM base mesh, which consists of position, color, texture coordinates and normals. Next, we construct data arrays for the user specified triangle type (disjoint, tstrips, indexed tstrips, indexed disjoint) in a way that data for the entire mesh lives inside a single array. When using VAs, there are separate arrays for vertex positions, normals, colors, etc. When using VBOs, a single array contains position data followed by color data followed by normal data, etc. This approach allows us to make on call to draw arrays rather than several calls, but requires the use of degenerate triangles in the tstrip representations.

A separate code `wesBench-vbo` contains the implementation that uses VBOs. Except for that minor difference, the codes are identical.

Both are written in ANSI C and should compile and run on any machine that supports OpenGL 1.2 for the VA code or OpenGL 1.5 for the VBO code.

8 Appendix C – Quadro FX 4500 Performance

For this study, we used an NVIDIA Quadro FX 4500 graphics accelerator. The card is a couple of years old: a cursory search at `nvidia.com` turned up little information about the vendor's published geometry and fill rates for this card: they seem to only post specifications for current products.

A Google search turns up a couple of items:

1. A *Tom's Hardware* article[9] indicates: 181 Mtris/second and 11.3 Btexels/sec.
2. An online shopping page[10] indicates: 181 Mtris/second and 10.8 Btexels/sec.

We'll assume that this card is capable of 181M vertex operations per second and about 11B pixel operations per second.

[9]http://www.tomshardware.com/news/nvidia-announces-quadro-fx-3450,1234.html
[10]http://www.alibaba.com/product-free/12271733/Nvidia_Quadro_Fx_4500_512mb_Gddr3.html