OpenGL's Immediate Mode Interface on Open-Source Platforms

Ian D. Romanick
IBM
idr@us.ibm.com

Abstract

The immediate mode vertex interface in OpenGL is at a crossroads. The performance of numerous applications with large installed user bases depend heavily on the performance of the immediate mode interface. A change of just a few clock cycles for each entry point can be felt by users of these applications. At the same time, many newer applications do not use the immediate mode interface at all, and some newer versions of OpenGL (e.g., OpenGL ES) have removed the interface altogether. This paper presents one possible path forward that addresses the two disparate goals of deprecating the immediate mode interface and making it high performance.

1 Introduction

Backwards compatibility is one of the foremost tenants of OpenGL. Programs written to the OpenGL 1.0 specification published in 1992 will still function correctly on OpenGL 2.0 implementations, some fourteen years later. This feat has eased the maintenance of numerous large-scale applications.

However, this has come at a cost. OpenGL implementations need to implement no less than *four* different interfaces for submitting vertex data. This is a heavy burden for driver writers. Each interface must be implemented, tested, and optimized separately. This complicates the code and testing of a driver.

The plurality of interfaces is also a burden for application developers, particularly novice OpenGL developers. With so many options available for an operation so fundamental as submitting vertex data, how is a developer to know which interface is right for a particular application? Moreover, how is a developer to know which interface is optimal on a particular implementation?

The end result is that very similar applications will have wildly different interface usage patterns. This presents additional difficulties for the driver writer. It is nearly impossible to guess what usage pattern applications will assume is the fast-path. With a few notable exceptions, driver writers are forced to guess which paths need the most optimization. This results in some usage patterns being optimal on some drivers and other usage patterns being optimal on other drivers. This makes it even more difficult for application developers to know which are the fast-paths, and the cycle perpetuates.

For these reasons, Khronos eliminated some older interfaces from OpenGL ES. At the top

of their list was the immediate mode vertex interface. In addition, the OpenGL ARB is debating the removal of these interfaces from a future version of OpenGL[1]. It has been proposed that the removed interfaces can be implemented via a shim between the application and the OpenGL implementation. This shim would convert the eliminated interfaces to more modern equivalents.

The remainder of this paper is divided into five sections. Sections 2 and 3 describe the current implementation of the immediate mode interface shipping in X.org 7.0 and a possible reimplementation within libGL. Section 4 presents projected performance impacts of such a reimplementation. Section 5 presents some extensions that could improve the performance of the reimplementation, and section 6 concludes the paper.

2 Current Implementation

In X.org's libGL, each OpenGL function is implemented in a manner similar to a C++ virtual function. Each GL context has an associated function table called a *dispatch table*. Each time an application calls an OpenGL function, it actually calls into a short stub function, called a *dispatch function*, that redirects to the true function based on a pointer stored in the dispatch table.

The dispatch implementation makes heavy use Linux's *thread local storage* (TLS) mechanism. Each thread has a per-thread variable called `_glapi_Dispatch_tls` that stores a pointer to the current dispatch table. Figure 1 shows the dispatch function for `glVertex3fv`. On most platforms, the compiler is able to optimize these functions to a couple pointer fetches and a single branch instruction.

```
glVertex3fv( const GLfloat * v )
{
   (*_glapi_Dispatch_tls->Vertex3fv)(v);
}
```

Figure 1: TLS optimized dispatch function

There is measurable overhead caused by the indirection. Applications that make heavy use of the immediate mode interface for vertex submission and are not fill rate bound, may see dispatch overhead as a performance bottleneck. Applications that commonly fit this profile are CAD applications and visualization applications used by the oil and gas industry. Unfortunately, these are also the same applications whose developers are most reluctant to move their existing code base away from the immediate mode interface.

This presents an unfortunate situation. The applications that most need a compatibility shim are also the ones that will see the largest performance degradation cause by any overhead added by the shim.

3 Immediate Mode Emulation

libGL is currently designed to provide a very thin layer between the application and the driver. The conventional wisdom is that having a thin layer minimizes the overhead imposed by the layer. The unexpected side-effect is that by having the layer be *too* thin, the layer always adds overhead between the application and the driver. By having the layer be a little thicker in some places, the overhead can be minimized further or eliminated altogether in other places.

The immediate mode interface can be implemented directly in libGL. Regardless of the

[1]The debate is *when*, not *if*.

driver loaded, libGL would store incoming vertex data in internal buffers. When `glEnd` is called, the data would be flushed to the driver. This technique, which will be explained further in section 3.1, even applies to indirect rendering.

There are two significant advantages to implementing this emulation layer in libGL. Since all of the code is in libGL, the overhead imposed by the dispatch layer is eliminated. For applications that are dispatch bound, this could actually improve performance. In addition, moving this code into libGL means that driver developers don't need to maintain these code paths in their drivers.

Platforms that don't want to support the immediate mode interface (e.g., OpenGL ES), can provide a libGL that omits that functionality. No modifications to the drivers would be required.

3.1 Implementation

This section provides an overview of a possible reimplementation of the immediate mode interface directly within X.org's libGL. While this is a brief overview, some important, difficult to handle cases are covered in detail.

Each immediate mode entry point is implemented in libGL. With the exception of provoking functions such as `glVertex3f`, each immediate mode function simply stores its input data in a per-thread buffer and returns. When a vertex is provoked, the data in each of the per-thread buffers is copied to the next element in a vertex array. When the arrays are filled or `glEnd` is called, the data is passed from libGL to the driver using the standard vertex array interface.

In an ideal implementation, all vertex data is stored in buffer objects and drawn with a single `glDrawArrays` call. However, several legal usage patterns prevent this ideal case from always being possible.

In addition to all the immediate mode data functions, `glBegin` and `glEnd` are completely subsumed by libGL. Since all functions that are valid within a `glBegin` / `glEnd` pair are handled directly by libGL, once `glBegin` is called the driver cannot be directly called. libGL must replace the current dispatch table with a special table of functions that set an error of `GL_INVALID_OPERATION` and return. This eliminates all possibility of `glBegin` / `glEnd` errors within the driver. All related error checking code can then be removed from the driver.

3.1.1 Non-array Data

For the most part, the only functions allowed within a `glBegin` / `glEnd` pair are functions that set vertex data. There are a few notable exceptions that must be handled specially. For example, if `glMaterialf` is called within a `glBegin` / `glEnd` pair, all buffered data must be submitted to the driver before calling the driver's `glMaterialf` function.

3.1.2 Non-uniform API Usage

Several classes of non-uniform API usage are allowed by OpenGL but are difficult to handle. Developer documentation frequently advises that vertex data be submitted in identical groups with the same type [2] [5] [7], but many applications exist that do not follow this advice. Some examples of this pattern include:

- Making immediate mode function calls outside a primitive (see figure 2).

```
glColor3fv( curr_color );
glBegin( GL_TRIANGLES );
for ( i = 0 ; i < num_vert ; i++ )  {
   glNormal3fv( normals[i] );
   glVertex3fv( vertices[i] );
}
glEnd();
```

Figure 2: Setting data outside a primitive

```
glBegin( GL_TRIANGLES );
glVertex3fv( v1 );
glVertex3iv( v2 );
glVertex3dv( v3 );
glEnd();
```

Figure 3: Mixing data types within a primitive

- Mixing data types of a particular element within a primitive (see figure 3).
- Mixing data counts of a particular element within a primitive (see figure 4).
- Using different data per-vertex within a primitive (see figure 5).
- Mixing immediate mode calls and calls to either `glArrayElement` or `glDrawElements`.

Mixing immediate mode calls and `glDrawElements` deserves further comment. Developers of modern OpenGL applications are known to use this technique to implement geometry instancing [6]. The OpenGL ARB recently decided that a dedicated geometry instancing interface was not needed for OpenGL [1]. It was believed that a dedicated interface would provide little performance or functional benefit over what can already be achieved using the immediate mode interface in conjunction with vertex arrays. As a result, modern applications contain code similar to figure 6.

```
glBegin( GL_TRIANGLES );
glVertex3f( x1, y1, z1 );
glVertex2f( x2, y2 );
glVertex4f( x3, y3, z3, w3 );
glEnd();
```

Figure 4: Mixing data counts within a primitive

```
glBegin( GL_TRIANGLES );
glNormal3fv( n );
glColor3fv( c1 );
glVertex3fv( v1 );
glVertex3fv( v2 );
glColor3fv( c3 );
glVertex3fv( v3 );
glEnd();
```

Figure 5: Mixing data elements within a primitive

3.1.3 Display Lists

Display lists provide a completely flexible way to record a series of OpenGL commands that can be replayed at a later time. The extreme flexibility has allowed application to developers to use, some would say *mis*use, display lists in ways not envisioned by its creators. This has made it very difficult for driver writers to optimize display lists. Since most drivers do not optimize display lists to their fullest potential, most application developers no longer use display lists for their intended purposes. Vertex arrays and buffer objects now provide a much higher performance path. Instead, only the ill-behaved uses are commonly used by applications.

Two common misuses of display lists cause the greatest difficulties for driver writers and for the immediate mode emulator. Display lists can contain mismatched `glBegin` / `glEnd` pairs or no `glBegin` / `glEnd` pair at all. While this may seem like an odd usage, many applications

```
for ( i = 0 ; i < num_instances ; i++ ) {
   glMultiTexCoord4fv( GL_TEXTURE0,
               transform[i][0] );
   glMultiTexCoord4fv( GL_TEXTURE1,
               transform[i][1] );
   glMultiTexCoord4fv( GL_TEXTURE2,
               transform[i][2] );

   glDrawElements( GL_TRIANGLES, num_indices,
                   GL_UNSIGNED_INT, indices );
}
```

Figure 6: Mixed arrays and immediate mode to implement instancing

contain code similar to figure 7.

Display lists can also contain incomplete vertex definitions, such as in figure 8). While this usage is much less common, it is known to exist in shipping applications. It presents similar implementation difficulties.

In a twist of cruel irony, a way to handle both these uses exists by further misusing OpenGL functionality. Figure 9 and figure 10 are functionality equivalent[2].

The technique in figure 10 can be used to send extra data to the driver when `glEndList` is called. The unfortunate side effect is that libGL must also track all array enables and disables so that the proper array state can be set.

3.2 Vendor Extensions

Some OpenGL extensions, such as `GL_ARB_vertex_blend`, add new immediate mode entry points. Only one such extension currently exists that is not already supported by libGL. It is unlikely that additional such extensions will be added in the future. The best solution will likely be to add support to libGL for the one missing extension and close the issue.

[2]The proper array enables must be set in the vertex array case.

```
glNewList(l, GL_COMPILE);
glVertex3fv(v[0]);
glVertex3fv(v[1]);
glVertex3fv(v[2]);
glEnd();
glEndList();

glColor3fv(solid_color);
glBegin(GL_TRIANGLES);
glCallList(l);

glColor3fv(line_color);
glBegin(GL_LINE_LOOP);
glCallList(l);
```

Figure 7: Mismatched `glBegin` / `glEnd` in a display list

```
glNewList(l, GL_COMPILE);
glNormal3fv(n[0]);
glColor3fv(c[0]);
glVertex3fv(v[0]);
glNormal3fv(n[1]);
glColor3fv(c[1]);
glEndList();
```

Figure 8: Incomplete vertex definition in a display list

With the implementation proposed in section 3.1 it is not possible with the emulator for a driver to add support for a new immediate mode entry point. The immediate mode entry points in libGL will store data into an array and advance the array pointers at each call to a provoking function (e.g., `glVertex3fv`). However, the driver never sees the provoking function calls. It, therefore, never knows when to capture the data or advance the array pointers.

Since it is improbable that additional immedi-

```
glColor3fv(c[0]);
glNormalfv(n[0]);
```

Figure 9: Setting color and normal via immediate mode interface

```
glColorPointer(3, GL_FLOAT, 0, c[0]);
glNormalPointer(GL_FLOAT, 0, n[0]);
glArrayElement(0);
```

Figure 10: Setting color and normal via array interface

ate mode entry points will be added by vendors or the ARB, this is unlikely to be a problem in practice. If this is becomes a problem, there are two viable options. Support for the new functions can be added to libGL. This is probably the best choice.

Alternately, a new function could be provided by the driver that libGL can use to signal it to advance its internal array index. This may be easier to implement in the short term, but it seems more likely to end up as cruft that libGL will have to carry around for little or no benefit to most drivers.

For now the answer is to cross this bridge when we come to it.

4 Projected Performance

To test the performance impacts, a simple wrapper was created that converts vertex, normal, and color immediate mode calls into vertex arrays.

In some ways, the wrapper should be considered the best case scenario. There are a couple caveats to consider. By removing support for the immediate mode interface from the driver, new optimization opportunities will present themselves (e.g., removal of all `glBegin` / `glEnd` tests). Clearly, the wrapper does not take any of these optimizations into consideration. All tests with the wrapper were performed with unmodified drivers.

The wrapper is configurable, at compile time, to use either "classic" vertex arrays or buffer objects. In addition, two different buffer object modes can be utilized. In the first mode, the same buffer objects are repeatedly filled with new data and reused. In the other mode, a new set of buffer objects are created for each batch of data. The buffer object specification was created to encourage this sort of "fire and forget" model when streaming dynamic data into OpenGL [8]. This *should* be the optimal usage model for the wrapper.

4.1 Test Setup

All tests were performed on an Athlon64 3000+ system. A system with a slow processor was used to exaggerate the overhead incurred by the wrapper. The wrapper does not handle any of the difficult cases mentioned in the previous section. To further exaggerate the CPU overhead, all rendering was done to a 50-by-50 window.

Tests were performed on two different graphics cards, each with two different drivers. A Radeon 8500LE and a Radeon 9600XT were both tested with the open-source drivers from Mesa CVS and with ATI's closed-source fglrx drivers. These cards and drivers were chosen because they each use a similar dispatch mechanism. This allowed certain optimizations in the wrapper, and these optimizations allowed the wrapper to more closely model a true implementation inside libGL.

Nvidia hardware was not considered for this test. Nvidia hardware, while common on

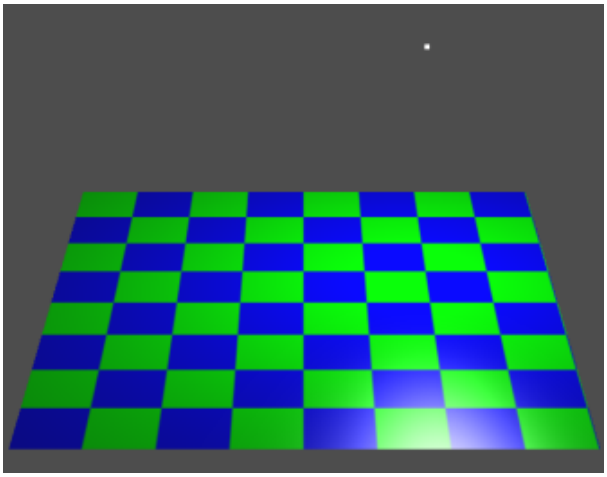

Figure 11: Sample image from test program

systems that use X.org, is only supported by closed-source drivers supplied by Nvidia. Nvidia's drivers use a very different dispatch mechanism than either the open-source drivers or the fglrx drivers. These differences caused any comparisons between the wrapped and non-wrapped interfaces to be uninteresting.

The test program renders a checkerboard with a single moving light source. The checkerboard is tessellated to 256-by-256 quadrilaterals. Figure 11 shows a sample frame. The test can submit from one to 256 quadrilaterals in each batch (e.g., within one `glBegin` / `glEnd` pair). Each test configuration renders the same total number of polygons, so performance comparisons between each batch size are possible. A more useful comparison is between different rendering paths at the same batch size.

4.2 Results with Open-Source Drivers

At small batch sizes, the overhead associated with the wrapper is very harmful to performance. Figures 12 and 13 both show the same trend. Batches of 8 or fewer polygons show dramatically reduced performance.

For batches of 16 or more polygons, performance is at least as good as the baseline. For batches of 32, 64, and 128 polygons, the wrapper *improves* performance. This is especially startling since the open-source drivers for the Radeon family of GPUs all have hand-optimized immediate mode paths.

On the Radeon 8500LE, the "fire and forget" buffer object path showed very poor performance. As shown in table 2, this path was, at best, only able to achieve 10% of the performance of the other paths. In the worst case it achieved only 0.03% of the performance of the baseline path. Since this is the manner in which buffer objects are intended to be used for dynamic, streaming data, this shows a clear deficiency in Mesa's buffer object implementation.

The picture on the Radeon 9600XT is somewhat different. Here the "fire and forget" path performed moderately. The reused buffer object path gave the worst performance of all tested configurations. From table 4, this configuration loses 87 frames per second from the baseline configuration. The reused buffer object path likely gives poor performance due to synchronization overhead with the GPU caused at each call to `glMapBuffer`.

4.3 Results with fglrx Drivers

As expected, increasing the batch size, in all configurations of the wrapper, increased performance. The buffer object configurations showed a linear doubling of performance as the batch size doubled.

Interestingly, the performance of the vertex array configuration peaks at a batch size of 64 polygons. Beyond that size the performance gradually falls off. This is especially surprising since the performance of the non-wrapped configuration continues to increase with batch size.

The most shocking data is the raw performance of the buffer object path on the fglrx drivers. The "fire and forget" buffer object path tops out, with a 256 polygon batch size, at 3% of the baseline. The percentage of baseline is much worse than the open-source drivers, but the raw frame rate is comparable.

The buffer object path that reused buffer objects fared slightly better, but the results are still disappointing. At each batch size this configuration was approximately 20% faster than the other buffer object configuration. Even with this 20% improvement, the open-source drivers achieve nearly eight times the throughput for 256 polygon batches.

It is possible that the fglrx drivers implement `glMapBuffer` by creating a shadow buffer in system memory. When `glUnmapBuffer` is called, the contents of the shadow buffer is copied to on-card memory. In the 16 polygon batch case, this means that 128KiB of data are copied when less than 2KiB of data is actually used. Reducing the buffer object size used by the wrapper from 128KiB to 9KiB increased performance by a factor of 4 for 256 polygon batches.

5 Future Extensions

Opportunities exist to create additional OpenGL extensions that could improve the performance of the immediate mode emulator. These extensions could also replace some of the functionality lost by removing the immediate mode interface.

5.1 True Zero Vertex Stride

Moving redundant operations outside a loop is well known optimization. OpenGL programmers are also familiar with this optimization, and they often use by moving redundant data outside a `glBegin` / `glEnd` pair. Data is then uniformly set inside the pair. Figure 2 shows a typical example. Figure 6 carries the identical issue, but uses `glDrawElements` instead of an explicit `glBegin` / `glEnd` pair.

To support a case like figure 2, libGL would have to make `num_vert` copies of the specified color value to pass to the driver. This is very inefficient, especially for large data sets. Two similar ways exist to address this problem.

The vertex array interface can be extended to accept a special stride value that instructs the driver to not advance the pointer for each element. Specifying a stride value of zero seems to be the obvious choice, but that value already has a special meaning in OpenGL. Using either -1 or a negative value equal in magnitude to the element size (e.g., -12 for an array with a size of 3 and a type of `GL_FLOAT`) would also work. If -1 is chosen, a special enumerant such as `GL_NO_STRIDE` should be created.

While it is clear how such an approach would interact with `glDrawArrays`, it is not clear how it would interact with element oriented drawing commands. It is also unclear whether such an extension would have any use outside the emulator.

Alternately, a more full featured instancing interface can be implemented. One possible implementation of such an interface adds per-array state that dictates the frequency with which the array pointer is updated[4]. Such an interface, if it supports fixed-function attributes, would improve the performance of the emulator.

A full featured instancing interface would also eliminate one of the most common cases where modern applications use the immediate mode interface. This eliminates part of the need for the emulator!

5.2 Array State Containers

Several corner-cases require array state to be disabled, pointers changed, and new array state enabled. These state changes, which are expensive enough on their own, are bracketed by calls to `glPushClientAttrib` and `glPopClientAttrib`. Much of the expense of these operations and complexity of the code could be eliminated by encapsulating array state in an object[3].

The emulator would no longer need to jump through numerous hoops, including the tedious operation of disabling all arrays. It would simply bind a new array state object, do its work, and rebind the old array state object. Not only would these operations be easier to implement and more performant, they would be much less likely to contain bugs.

5.3 Disable All Arrays

A fair amount of work is required by the driver writer to implement the additional functionality proposed in section 5.2. Some of the benefit gained by that extension could also be gained with a much simpler extension. By adding a new enumerant to `glDisableClientState` that disables *all* previously enabled arrays, the implementation of the emulator could be simplified.

5.4 Flush Callback

A common optimization in OpenGL drivers is to merge batches primitives from multiple `glBegin` / `glEnd` pairs together. If an application sends multiple small batches of primitives without intervening state changes, the driver can treat the individual batches as a single, large batch. Without knowledge of potential state changes between a call to `glEnd` and the next call to `glBegin`, this optimization is impossible.

A simple extension enables the driver to communicate this information back to the emulator. Using the extension, libGL registers a callback function with the driver. When driver detects that a state change had occurred, it invokes the callback function. libGL then flushes any pending drawing commands to the driver. Internally, Mesa already implements a nearly identical mechanism.

Implementing such a mechanism for indirect rendering should trivial. Indirect rendering implementations already combine multiple immediate mode calls into large batches. These batches are sent in a single Render protocol packet. When a state change function is called, the pending Render packet is sent to the server before sending the state change packet.

Such a callback mechanism is a complete moral violation of the OpenGL design. The potential performance benefits for applications that make heavy use of the immediate mode interface should out weigh any qualms that a driver writer may have.

5.5 Subrange Unmap

The performance anomalies with buffer objects shown in section 4.3 may be tracable to inherrent inefficencies with `glUnmapBuffer`. The problem, which could easilly apply to applications as well, is `glUnmapBuffer` must assume that every byte of the buffer object has been modified. If only a small portion of a large buffer object has been modified, this may result in large amounts of data being needlessly copied from system memory to on-card memory.

An extension that allows an application or the immediate mode emulator to mark subregions

of the buffer object as "dirty" would eliminate this problem. This trivial extension could take either of two forms. A special version of `glUnmapBuffer` could be added that takes a list of dirty ranges as a parameter. Alternately, a new function could be added that allows applications to mark regions of the buffer as dirty *before* calling `glUnmapBuffer`.

6 Conclusion

Fully implementing OpenGL's immediate mode interface as a layer on top of the existing vertex array interface is a large, complex task. The non-obvious ways in which applications can combine the immediate mode interface with display lists and vertex arrays creates a number of corner cases that are likely to be more common in practice than one would hope. Unfortunately, handling these cases correctly is not enough. High performance, relative to existing "native" implementations of the immediate mode interface is also required.

Drivers internally utilize extra data to perform additional optimizations when the immediate mode interface is used. The majority of this data, such as whether or not a state change has occurred, is simply not available to a layered implementation. Implementing a communication interface, perhaps via OpenGL extensions, between libGL and the driver may make some of these optimizations available.

Implementing such extensions has two potential disadvantages. First, implementation of these extensions will nullify some of the code reduction benefits of moving the immediate mode interface code from the driver to libGL or a layered library. In addition, implementing these extensions creates a tighter coupling between libGL and the driver. Some driver writers may find this objectionable.

The immediate mode interface provides functionality that is not available through other means. If a future version of OpenGL is to truly deprecate the immediate mode interface, suitable replacements must be provided. It is likely that extensions can be created that will suit the needs of both users and the immediate mode wrapper.

Data clearly shows that an immediate mode wrapper will cause a performance hit. Vendors of applications whose performance is tied to the immediate mode interface need to begin considering their available options and working with driver and hardware vendors *now*.

Not only is this as wake-up call for application developers but also for driver writers. The poor performance of buffer objects on all tested card and driver combinations is cause for further alarm. Any optimal immediate mode emulator will need to use buffer objects. Until changes are made within the available drivers, this will not be a performant option.

References

[1] OpenGL ARB meeting minutes, December 2004. `http://www.opengl.org/about/arb/notes/meeting_note_2004-12-07.html`.

[2] David Blythe, Brad Grantham, Mark J. Kilgard, Tom McReynolds, and Scott R. Nelson. Advanced Graphics Programming Techniques Using OpenGL. In *SIGGRAPH '99 Conference Proceedings*. SIGGRAPH, ACM, August 1999. `http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/`.

[3] APPLE_vertex_array_object extension specification, 2002. `http:`

//oss.sgi.com/projects/
ogl-sample/registry/APPLE/
vertex_array_object.txt.

[4] Simon Green. NVIDIA OpenGL update, 2006. http://
developer.nvidia.com/object/
opengl-nvidia-extensions-gdc-2006.
html.

[5] Hewlett Packard. *HP's Implementation of OpenGL: HP 9000 Workstations*, 1997.

[6] Matt Pharr. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[7] Sun Microsystems. *Sun OpenGL 1.2.1 for Solaris Implementation and Performance Guide*, 2000.

[8] Ian Williams and Evan Hart. Efficient rendering of geometric data using OpenGL VBOs in SPECviewperf. Technical report, June 2005. http://www.spec.org/gpc/opc.
static/vbo_whitepaper.html.

Config. / Quads per batch	1	2	4	8	16	32	64	128	256
Baseline	28.2	45.4	54.4	60.6	64.2	66.2	67.3	68.2	68.5
Vertex Array	6.1	11.7	22.3	39.4	63.5	65.4	66.7	67.2	65.4
Buffer Object	0.02	0.05	0.09	0.2	0.4	0.8	1.5	3.0	5.8
Reused Buffer Object	0.03	0.05	0.1	0.2	0.4	0.8	1.6	3.2	6.2

Table 1: Frame rates for Radeon 8500LE (fglrx)

Config. / Quads per batch	1	2	4	8	16	32	64	128	256
Baseline	17.6	24.3	32.3	36.0	41.0	43.6	45.5	46.2	46.6
Vertex Array	2.2	4.1	13.8	23.5	37.1	49.1	49.1	49.4	49.3
Buffer Object	0.03	0.05	0.1	0.2	0.4	0.8	1.6	3.0	5.7
Reused Buffer Object	2.0	3.7	11.5	21.5	33.9	47.4	49.1	49.5	49.3

Table 2: Frame rates for Radeon 8500LE (Xorg)

Config. / Quads per batch	1	2	4	8	16	32	64	128	256
Baseline	54.5	81.8	108.1	127.8	140.8	149.8	154.9	158.3	162.0
Vertex Array	17.6	32.4	55.7	83.6	106.1	117.0	120.9	119.6	109.9
Buffer Object	0.02	0.05	0.1	0.2	0.4	0.8	1.6	3.0	5.6
Reused Buffer Object	0.03	0.07	0.1	0.3	0.5	1.1	2.1	3.9	7.2

Table 3: Frame rates for Radeon 9600XT (fglrx)

Config. / Quads per batch	1	2	4	8	16	32	64	128	256
Baseline	14.7	26.0	43.0	64.5	75.6	80.3	85.4	88.3	89.2
Vertex Array	1.3	2.7	5.3	10.3	19.6	35.6	59.5	92.2	108.9
Buffer Object	0.9	1.8	3.5	6.5	11.4	18.1	25.5	31.5	35.6
Reused Buffer Object	0.01	0.02	0.03	0.06	0.1	0.2	0.5	1.0	2.0

Table 4: Frame rates for Radeon 9600XT (Xorg)

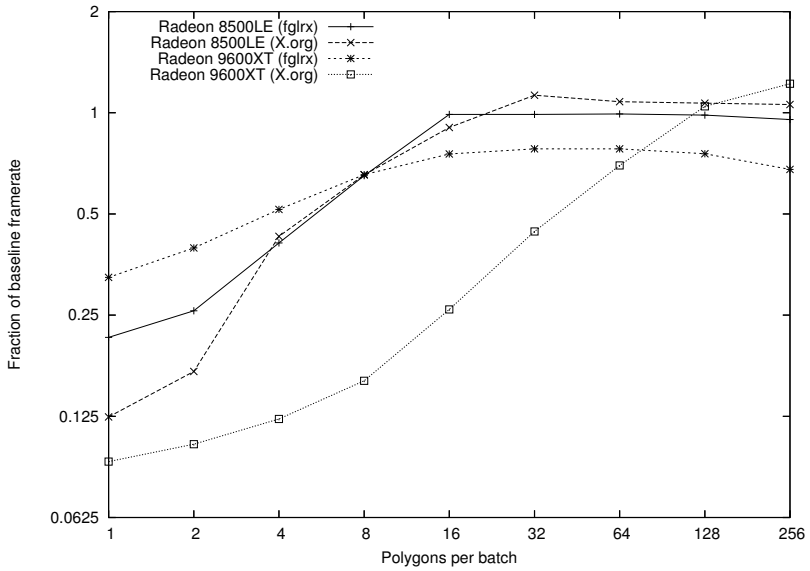

Figure 12: Performance using vertex arrays

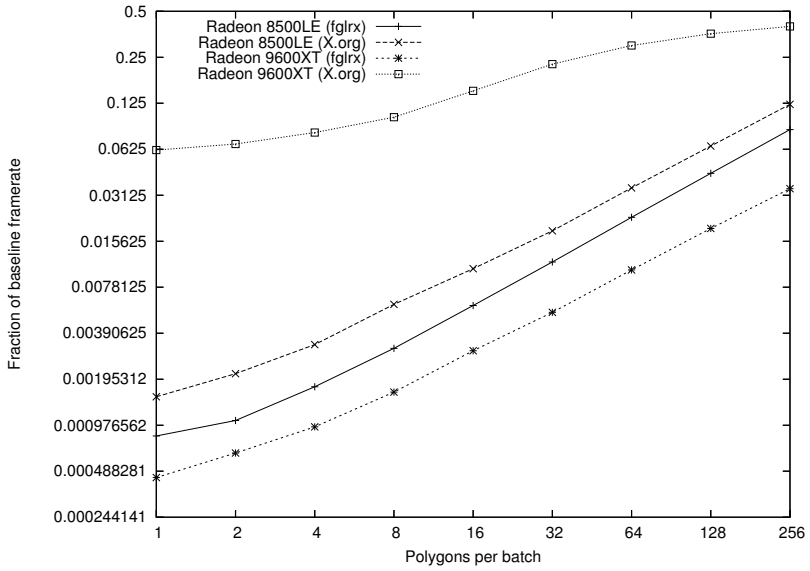

Figure 13: Performance using buffer objects

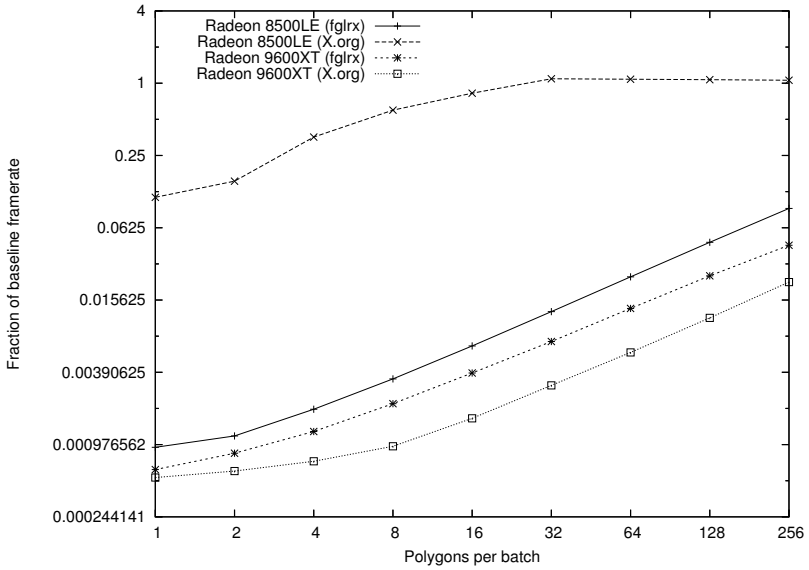

Figure 14: Performance using reused buffer objects