

# Improving Performance by Reducing Calls to the Driver

25

Sébastien Hillaire

## 25.1 Introduction

Rendering a scene can involve several rendering passes such as shadow map construction, light contribution accumulation, and framebuffer postprocessing. OpenGL is a state machine, and each of these passes requires changing state several times. Rendering requires two main steps:

1. Modify OpenGL states and objects in order to set up the assets used for rendering.
2. Issue a draw call to draw triangles and effectively change some pixel values.

These two steps require multiple calls to the driver. The driver is responsible for translating these function calls into commands to be sent to the GPU. The driver is provided by the GPU vendor and could be considered as a black box where only the vendor knows what is being done by each function call. At first, it seems reasonable to think that the drivers are filling a FIFO command queue that will be used to render the next frame. However, the driver's behavior can be very different depending on the vendor and/or platform. For example, a GPU driver for desktop computers has to take into account the wide variety of hardware that makes drivers a lot more complex than their counterpart on consoles, for which the platform is entirely and reliably known [Carmack 11]. Indeed, on such a platform, the driver can be specifically optimized for the installed hardware, whereas on the PC, the API requires a higher level of abstraction. Therefore, we should assume that each call to the OpenGL API

will result in costly driver operations such as resource management, current state error checking, or multiple shared context threads synchronizations.

This chapter presents solutions that can be used to reduce the number of calls to the graphic driver in order to improve the performance. These solutions allow us to reduce the CPU overhead and hence increase rendering complexity.

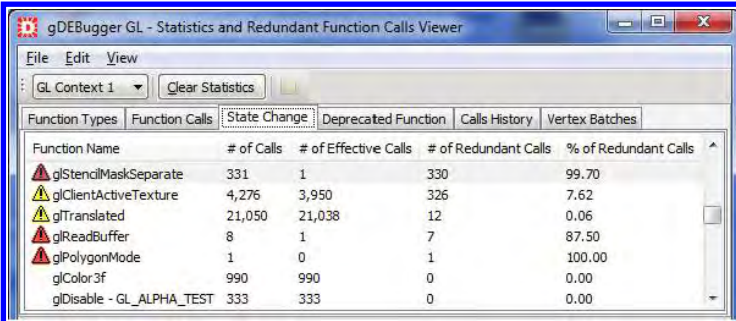
## 25.2 Efficient OpenGL States Usage

Accessing and modifying OpenGL states can only be done through multiple calls to the API functions. Each call can potentially consume a lot of processing power. As a result, care must be taken to efficiently change OpenGL states using as few API calls as possible. In the OpenGL 1.0 days, state change operations could be accelerated using display lists which stored precompiled commands that could be executed in a single call. Despite being static, display lists were used as a fast way to change OpenGL states. However, they were removed from the OpenGL 3.2 core profile but are still available in the compatibility profile.

This section presents ways of detecting and avoiding unnecessary API calls. We also present recent OpenGL features that allow us to increase the efficiency of each call intended to change the current OpenGL states.

### 25.2.1 Detecting Redundant State Modifications

Debugging and optimizing an OpenGL application can be made easier by using some specific existing software. *gDEBugger* [Remedy 11] is a free tool that can record the OpenGL function call sequence for each frame independently. An important feature of this software is the *Statistics and Redundant Function Calls Viewer*, as seen in Figure 25.1, accessible using Ctrl+Shift+S. It allows us to count the number of



Function Name	# of Calls	# of Effective Calls	# of Redundant Calls	% of Redundant Calls
glStencilMaskSeparate	331	1	330	99.70
glClientActiveTexture	4,276	3,950	326	7.62
glTranslated	21,050	21,038	12	0.06
glReadBuffer	8	1	7	87.50
glPolygonMode	1	0	1	100.00
glColor3f	990	990	0	0.00
glDisable - GL_ALPHA_TEST	333	333	0	0.00

Figure 25.1. Using gDEBugger statistics to detect redundant OpenGL calls.

redundant calls to the driver that do not change its state. These calls are hence useless and should be avoided for the benefit of performance on the CPU/application side.

gDEDebugger can also be used to verify that no deprecated functions are called. One of its drawbacks is that it is currently limited to OpenGL 3.2. However, more runtime debugging possibilities are offered such as OpenGL data/states viewing, global statistics, and performance analysis.

### 25.2.2 General Methods for Efficient State Modification

To avoid redundant calls, a solution would be to rely on the `glGet*` functions in order to query an OpenGL state's value before each API call that might modify it. This approach must be avoided, as it is not efficient. This is due to the fact that the driver may have to look for the value resulting from previous commands appended to the command queue. Instead, two software solutions should be preferred: *return-to-default-state* or *state tracking*.

The return-to-default-state method is a straightforward one. OpenGL is first initialized to what is called the *default states*. When rendering is required, OpenGL states are modified. After draw calls corresponding to this state has been issued, default OpenGL states are restored so that others parts of the program start from the same point when modifying the default OpenGL states assumed. This widely used approach, often in old maintained projects and demos, has the advantage of avoiding checking all OpenGL states before changing only a few of them. The drawback is that it potentially issues twice as many API calls unless you rely on `glPushAttrib/glPopAttrib`, which is not a recommended approach.

The widely used alternative method called state tracking avoids redundant calls by keeping the OpenGL states up-to-date on the CPU side and keeping track of changes. This enables runtime evaluation of which driver calls are required to change the current states to the desired one. A very efficient implementation of this behavior is available in the game Quake 3 Arena [IdSoftware 05].<sup>1</sup> OpenGL states are kept in a single unsigned long value (Listing 25.1). Each bit of this double word stores whether or not an OpenGL state is activated. This enables the application to keep track of binary OpenGL states.

More complex states can also be tracked. As an example, the source and destination blend modes are kept in the lowest significant byte of the double word. If this byte is `0x00`, then blending is disabled, else the first and second hexadecimal are custom values representing the source and destination blend mode. When rendering the scene, each time a new material, e.g., shader, is selected to render surfaces, the `GL_State` function is called with the desired OpenGL states as a parameter (Listing 25.1). The differences as compared to current OpenGL states are first computed using XOR. This is used to check, for each OpenGL state, whether a change needs to be applied. If the result is not zero, then, the state needs to be changed and an

---

<sup>1</sup>See `GL_State` function in `tr_backend.c`.

```

void GL_State(unsigned long stateBits)
{
    // Xor operation to compute state that need to be changed
    unsigned long diff = stateBits ^ glState.glStateBits;

    // Check depthFunc bits
    if (diff & GLS_DEPTHFUNC_EQUAL_BITS)
    {
        if (stateBits & GLS_DEPTHFUNC_EQUAL)
        {
            glDepthFunc( GL_EQUAL );
        }
        else
        {
            glDepthFunc( GL_LEQUAL );
        }
    }

    // [Process other states...]

    // Store current state
    glState.glStateBits = stateBits;
}

```

**Listing 25.1.** Avoiding redundant OpenGL calls by storing current state on the CPU.

API call needs to be issued. At the end of this method, the current tracked state is replaced with the new one. The Quake 3 engine also tracks OpenGL objects that can be bound such as textures and the associated environment parameter for the first two texture units.

We have modified the Quake 3 rendering engine to compare the return-to-default-state approach to the state tracking one used in Quake 3. The performance was measured on a replayed session of the game with 32 bots playing against a human on the Q3DM7 map. The computer was equipped with an Intel Core i5 processor, 4GB of memory, and an NVIDIA GeForce 275 GTX. Results show that for a complex scene with surfaces ordered by material, the state tracking method is 11% faster than the return-to-default-state approach (Table 25.1). We must keep in mind that the difference would have been much higher on a computer from 1999 because of lower CPU frequency. Also, OpenGL now requires the use of shaders, uniform values, buffer objects, etc. These increase the number of states that must be tracked.

	Frames per second	milliseconds
states-tracking	714	1.4
return-to-default-states	643	1.55

**Table 25.1.** Mean performance measured when replaying 20 recorded sessions of Quake 3 arena with and without OpenGL states check on the CPU.

It is important to understand that the state-tracking method alone will not dramatically improve performance if meshes are rendered in a random order. It should only be seen as an additional component to more generalized optimization approaches. Indeed, the performance of an application will benefit more from the a priori knowledge we have when rendering meshes. For example, the Quake 3 engine sorts meshes per material so that they are drawn in a specific order, e.g., opaque, sky box, then transparent geometry. This helps us apply fewer changes to states related to material such as textures or alpha blending. Furthermore, state tracking can also be applied only when a material needs to be changed. You can also group some common states together to look for state differences at a coarse level first and then apply fine-grained state checks. Instead of material, grouping state changes as a function of their modification frequency can also be a good choice. To sum up, a priori knowledge, coarse state grouping, and fine-grained state changes are three methods that can be combined, or used independently, for efficient state tracking and modifications.

Another example of the use of such methods can be found in recent opensource game engines such as the engines used for games *Penumbra Overture* [Frictional-Game 10] and *Doom 3* [IdSoftware 11]. Unified lighting is achieved in the *Doom 3* renderer by using a common set of shaders to render all surfaces. As in Quake 3, an a priori knowledge is used to sort meshes according to their material. The state tracking method is also used to modify OpenGL states at specific stages of each code path for the unified lighting and shadowing methods. This reveals that these methods are timeless and should always be considered when developing any renderer.

## 25.3 Batching and Instancing

The performance of a rendering system will not only be driven by the number of triangles an application need to draw. The number of draw calls issued for each frame also plays a crucial role. Indeed, this is a very complex metric that depends a lot on the hardware [Wloka 03], i.e., CPU/GPU, and software, i.e., the driver [Hardwidge 03]. Thus, if CPU bound, the performance of an application will be mostly influenced by the number of batches per frame, a batch representing a draw call (`glDraw*`) that is often accompanied with state changes. If GPU bound, the performance will be influenced by the number of triangles drawn and pixel-shader complexity. Reducing the number of draw calls is mostly a CPU-only optimization. Thus, such reduction will not influence the performance of an application that is GPU bound because of complex geometry or shaders.

On a PC, the number of batches that can be submitted every frame is very limited when compared to consoles because of higher driver overhead [Hardwidge 03]. Wloka [Wloka 03] has shown that the number of batches that can be issued per frame highly depends on the CPU when there are few triangles because of the overhead resulting from the setup and commands submission to the driver (CPU-bound application). As a result, we can, to some extent, freely render more triangles per

batch without hurting the overall performance of the application. It shows that performance depends a lot on the CPU and GPU performance and the way they are tied together. Despite being old, the presentation of Wloka is a good starting point to begin understanding the cost of batches. We have to keep in mind that performance will be closely linked to our hardware and context of execution, and it can evolve with hardware. However, there are still some guidelines that can be followed to improve draw-call size and reduce the number of draw calls.

### 25.3.1 Batching

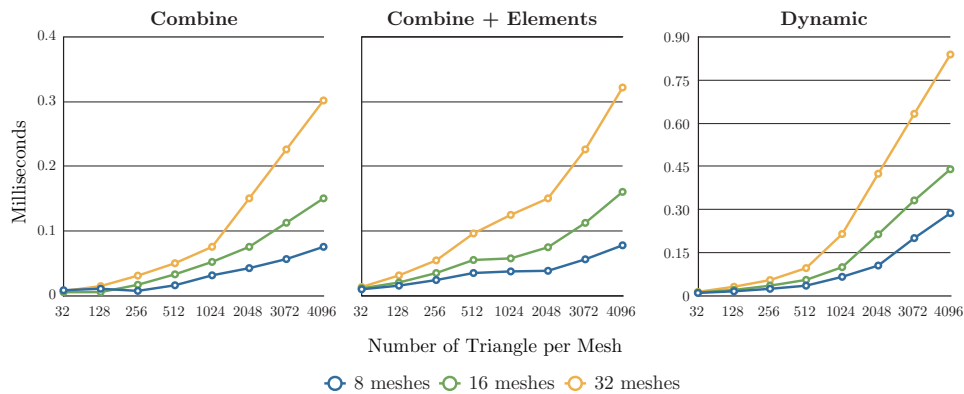
Batching refers to the general activity of grouping primitives together to render them all using as few draw call as possible. The larger the batches, the less the cumulative batch-submission overhead, i.e., fewer draw calls for fewer driver calls and CPU usage. Changes in transformation, material, or texture are the major batch breakers as these operations require changing OpenGL states. Several batching methods exist to reduce the number of draw calls: *combine*, *combine+element*, and *dynamic*.

The combine method packs together several geometry objects in a single set of buffer arrays (vertices and their attributes, indices) and renders them in a single draw call. The combination can be done based on object-appearance similarity. For example, small rocks could be combined and drawn together in a single draw call. The drawbacks are that objects can no longer move relative to each other, and culling will be limited. However, a certain amount of culling can be maintained if we take into account the relative position of objects in order to not pack together objects that are far away. You can also group together object that are in the same room or area of your scene.

The combine+element method consists of packing together geometry objects in a single set of buffer arrays (vertices and their attributes) but keeping the element array (indices) dynamic. As compared to combine, this approach allows us to have full control over object culling. It can be used for small-to-large static objects, as copying element indices can be done very fast. However, the combined array buffers containing geometry can take a lot of memory because they must contain all transformed geometry of all objects in the scene. The drawback of this method is that it can only be used to group nonanimated meshes.

The last method, dynamic, proposes to preallocate vertex and element buffers and to fill them dynamically at runtime. This approach can be used for objects sharing the same rendering state (shaders, textures, uniforms, blending, etc.) but potentially different vertex buffers over time, e.g., objects resulting from transformation or skinning. This approach is efficient for a lot of relatively small objects when computing vertex transformation on the CPU is faster than issuing more draw calls. Also, it takes some memory bandwidth to fill dynamic buffers, so performance tests must be conducted before choosing this method instead of issuing more draw calls.

In the case where different objects packed in a single array require different textures for their appearance, a texture atlas can be used. A texture atlas refers to a single



**Figure 25.2.** Batching performance for three different methods.

large texture containing several textures [NVIDIA 04]. Thus, changing appearance no longer requires changing the currently bound texture object. This method still suffers from the need to preprocess texture coordinates to match the texture atlas. Also, texture repetition cannot be used unless handled specifically in the fragment shader using more ALU operations. A more efficient approach on today's hardware would be to use texture arrays through the extension `ARB_texture_array`. In this case, a single texture object can address different textures of the same size according to a single index and without the need to process texture coordinates.

The performance of each method is presented in Figure 25.2 for 8, 16, and 32 meshes rendered. Performance is presented as a function of the number of triangles per mesh with each vertex attribute having 2-component texture coordinates and a 3-component normal. We can notice that the combine+element method has an overhead because it needs to send the element array before each draw call. That cost disappears when the number of triangles to draw increases. As expected, the dynamic method is the most costly one, as all data need to be sent to the GPU before each draw call.

The methods presented in this section effectively reduce the number of draw calls. However, they also require either a lot of preprocessing, making them harder to use in dynamically generated content environments. These solutions fit perfectly for most of the use cases. However, they would not be able to efficiently render a huge number of instances of an object with specific per-instance data. A solution for such a case is to use *instancing*.

### 25.3.2 OpenGL Instancing

The efficient rendering of a huge number of instances of objects having different positions and appearance is a complex task. With OpenGL, two methods can be used to achieve instancing.

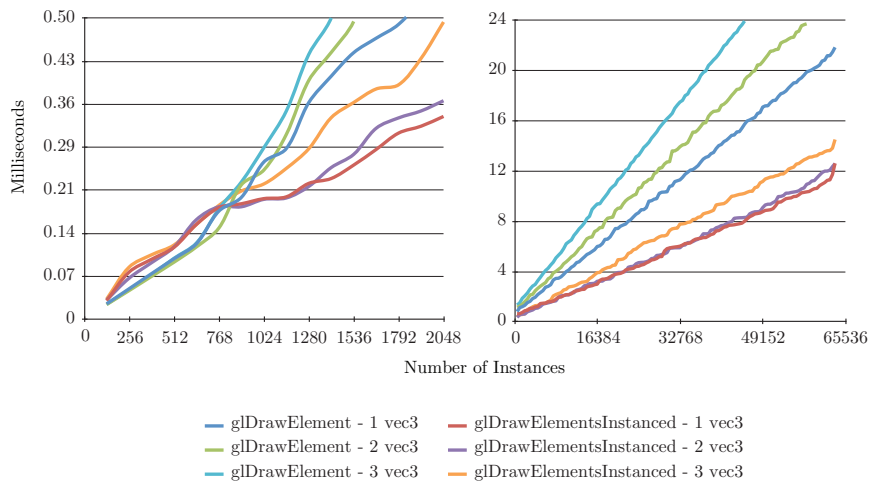
The first instancing method relies on the memory available through shader uniforms to store arrays of parameters. This array can be indexed through an additional per-instance vertex attribute. Thus, a large batch of geometry can be rendered in a single draw call, and each vertex will automatically read the input it needs to achieve a specific appearance and transformation in a way similar to indexed vertex skinning [Beeson 04]. With this method, we do not need any OpenGL extensions. The downside is that we are limited by the number of uniforms that can be allocated for each vertex shader. An extension to this method is to use vertex texture fetch to read data from textures. However, this requires having a graphic card supporting at least the OpenGL 2.0 core and, if we require floating point values, to check that the `GL_ARB_texture_float` extension is defined unless we use the OpenGL core 3.0. This method has the advantage of being usable on old hardware, but the number of instances drawn per draw call will be limited.

True instancing [Carucci 05] can be achieved with OpenGL 3.1 or through the `GL_ARB_instanced_array` extension. It is achieved using three specific steps: (1) bind array buffers to the attribute input of the vertex shader; (2) for each attribute input, specify the frequency at which vertex attributes need to be updated using the divisor value (attribute array pointer will be incremented every “*divisor*” instance); (3) call a draw function that takes as input the number of instances to draw, e.g., `glDrawElementsInstanced`. Additionally, the constant `gl_InstanceID` representing the number of instances already drawn is available in the shaders. It allows us to compute specific data for the current instance being rendered.

Concerning visual diversity, instancing is more limited than multiple independent draw calls. However, OpenGL extensions such as `ARB_base_vertex`, `ARB_base_instance`, or `ARB_texture_array` were designed to help developers restore that diversity on all instances, e.g., material parameters and textures.

To show the importance of instancing on GPU performance, I measured the cost of rendering different number of instances with a varying number of per-instance data. To get the raw performance, each instance consisted of a single triangle. I compared one draw call per instance, `glDrawElements`, to one single call for all instances, `glDrawElementsInstanced`. In the first case, per-instance parameters were sent through shader uniforms. In the second case, the divisor value is used with array buffers, as described previously, to get different input values for each instance (divisor = 1). Figure 25.3 presents the measured performance for several instances and per-instance vec3 parameters. All triangles were rendered outside the view frustum to avoid rasterization cost. Performance was measured on a Intel i5 processor, with 4GB of memory and a NVIDIA GeForce 275 GTX. This table reveals a cost overhead for the instancing method when there are fewer than 768 instances. Above that limit, the more per-instance parameters we add, the more interesting it is to use instancing. Also, 768 instances is a worst-case scenario as we are only instancing a single triangle. As revealed by Wloka [Wloka 03], we could also draw more triangles per instance without hurting performance: the more triangles per instance, the lower

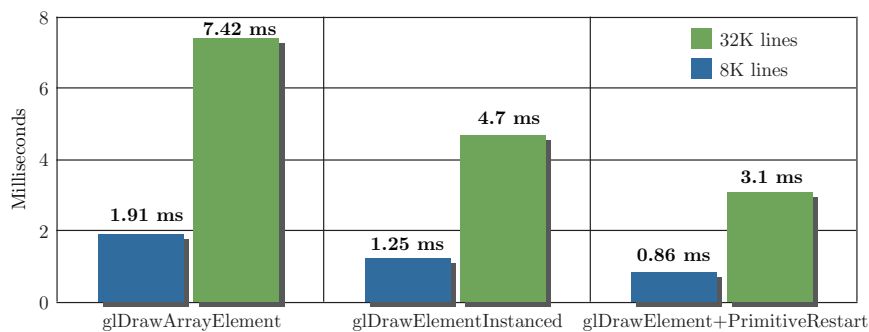




**Figure 25.3.** Performance comparison for one draw call per instance (`glDrawElements`) to one single call for all instances (`glDrawElementsInstanced`) for 1–3 per-instance `vec3` attributes.

the instancing initialization overhead. Also, the GPU rendering cost grows linearly with the number of instances drawn.

Instancing is an interesting method, but we must keep in mind that the cost overhead of each batch is much higher than a standard draw call. We have conducted another performance experiment when rendering volumetric lines filled with



**Figure 25.4.** Performance when rendering volumetric lines using three different approaches: one draw call per line, instancing, and multiple lines drawn using a primitive restart element.

a single color using the vertex extrusion method presented in Chapter 11. Performance is visible in Figure 25.4. In this case, the color is the parameter that needs to be modified for each line instance. As expected, the slowest method is again the one that issues a draw call for each line after having modified the uniform color value (Figure 25.4, `glDrawElement`). Using instancing, performance is 50% higher (Figure 25.4, `glDrawElementsInstanced`). In this case, an array buffer object is updated once per frame with colors of each instance. The color attribute is updated once per instance using the per-vertex attribute divisor. Surprisingly, the fastest method is the one relying on a dynamically filled array buffer object as illustrated in Chapter 11 (Figure 25.4, `glDrawElements+PrimitiveRestart`). In this implementation, the triangle strip vertices are generated on the CPU and sent into a vertex buffer object on the GPU. All strips are drawn using a pre-allocated element buffer that contains a special index value used to restart the strip primitives. This method indeed consumes a lot more memory than the one relying on instancing. However, because the cost of a primitive restart is negligible when compared to the overhead of using instancing, this method is actually the fastest. However, although Figure 25.4 does not show such a trend, we must keep in mind that the CPU might become the bottleneck when too many lines have to be processed. This last example shows that we must stay imaginative and try multiple approaches: the latest technology may not be the right choice in some cases and on some hardware. As in this example, the choice of an implementation can often be regarded as a quality/memory/computation complexity trade-off but not a technology decision.

## 25.4 Conclusion

Over the last several years, the OpenGL API has been heavily modified in order to make certain high-frequency calls more efficient for better overall performance. There is no doubt that upcoming years will reveal more API modification for the purpose of simplicity and efficiency. However, even with all these new tempting technologies, we must also keep in mind that traditional brute force approaches can still be faster under certain circumstances.

**Acknowledgments.** I would like to thank the editors, Randall Hopper, Aras Pranckevičius, Emil Persson, and an anonymous reviewer for their insightful comments during the review process of this article.

## Bibliography

[Beeson 04] Curtis Beeson. "Animation in the Dawn Demo." In *GPU Gems*. Reading, MA: Addison-Wesley, 2004.

[Carmack 11] John Carmack. "QuakeCon Keynote." QuakeCon Conference, Dallas, 2011.

- [Carucci 05] Francesco Carucci. “Inside Geometry Instancing.” In *GPU Gems 2*. Reading, MA: Addison-Wesley, 2005.
- [FrictionalGame 10] FrictionalGame. “Penumbra Overture Engine.” <http://frictionalgames.blogspot.com/2010/05/penumbra-overture-goes-open-source.html>, 2010.
- [Hardwidge 03] Ben Hardwidge. “Farewell to DirectX?” <http://www.bit-tech.net/hardware/graphics/2011/03/16/farewell-to-directx/1>, 2003.
- [IdSoftware 05] IdSoftware. “IdTech3 Source Code.” [http://en.wikipedia.org/wiki/Id\\_Tech\\_3](http://en.wikipedia.org/wiki/Id_Tech_3), 2005.
- [IdSoftware 11] IdSoftware. “IdTech4 Source Code.” <http://github.com/TTimo/doom3.gpl>, 2011.
- [NVIDIA 04] NVIDIA. “Improve Batching Using Texture Atlases.” [http://http.download.nvidia.com/developer/NVTextureSuite/Atlas\\_Tools/Texture\\_Atlas\\_Whitepaper.pdf](http://http.download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf), 2004.
- [Remedy 11] Graphic Remedy. “gDEBugger.” <http://www.gremedy.com>, 2011.
- [Wloka 03] Matthias Wloka. “Batch, Batch, Batch, What Does It Really Mean?” Game Developer’s Conference, San Francisco, 2003.