OpenGL Insights

Edited by

Patrick Cozzi and Christophe Riccio

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an informa business
AN A K PETERS BOOK

33 ARB_debug_output: A Helping Hand for Desperate Developers

António Ramires Fernandes and Bruno Oliveira

33.1 Introduction

Since the inception of OpenGL, error handling has not been without some controversy, as the only available mechanism to provide feedback was the `glGetError` function. For each OpenGL command, the application had to explicitly query for possible errors, getting in return a single and very broad meaning error identifier for the latest error.

`ARB_debug_output` [Konttinen 10a], originally proposed by AMD [Konttinen 10b], introduces a new feedback mechanism. It allows developers to define a callback function that will be invoked by OpenGL to report events back to the application. The callback mechanism frees the developer from having to explicitly check for errors during execution, populating the code with `glGetError` function calls. Nevertheless, the extension specification does not force the definition of a callback function. When no callback is defined, the implementation will keep an internal log called the message log.

The nature of the reported events is very broad and can relate, for instance, to errors using the API, the usage of deprecated functionality, performance warnings, or GLSL compiler/linker issues. Event information contains a driver-implementation dependant message and other data such as its severity, type, and source.

The extension also allows user-defined filtering of the events that are reported by selecting only the ones of interest. Finally, the application, or any third-party library, may also generate custom events. In the following sections, we will show how to use the aforementioned extension and its features. We will also have a peek at the current implementation approaches.

33.2 Exposing the Extension

As stated in the specification, it is recommended that the extension is available only in an OpenGL debug context to avoid a potential performance impact; hence, it may be required to create such a context. Using *freeglut* as an example, this can be achieved with `glutInitContextFlags(GLUT_DEBUG)`. An example of the flags to create an OpenGL 4.1 core debug context using `WGL` and `GLX` is presented in Listing 33.1. To check if the extension is available, we can use `glGetStringi`.

```
int attribs[] =
{
#ifdef WIN32
  WGL_CONTEXT_MAJOR_VERSION_ARB, 4,
  WGL_CONTEXT_MINOR_VERSION_ARB, 1,
  WGL_CONTEXT_FLAGS_ARB, WGL_CONTEXT_DEBUG_BIT_ARB,
  WGL_CONTEXT_PROFILE_MASK, WGL_CONTEXT_CORE_PROFILE_BIT_ARB,
#endif
#ifdef __linux__
  GLX_CONTEXT_MAJOR_VERSION_ARB, 4,
  GLX_CONTEXT_MINOR_VERSION_ARB, 1,
  GLX_CONTEXT_FLAGS_ARB, GLX_CONTEXT_DEBUG_BIT_ARB,
  GLX_CONTEXT_PROFILE_MASK, GLX_CONTEXT_CORE_PROFILE_BIT_ARB,
#endif
  0
};
```

Listing 33.1. Flags to create a core debug context in WGL and GLX.

33.3 Using a Callback Function

The extension allows the definition of a callback function that will be invoked each time an event is issued. This directs the flow of generated events to the callback, and they will not be stored in the message log.

If working with multiple contexts, each one should have its own callback function. Multithreading applications are allowed to use the same callback for multiple threads, and the application is fully responsible for ensuring thread safety.

Listing 33.2 is an example of such a callback function; it prints the event passed by OpenGL in a human-readable form. The enumerations are defined in the extensions spec [Konttinen 10a]. The `getStringFor*` functions translates the enumeration value into a human readable format. The complete code can be found on the book's web site (www.openglinsights.com).

The function `glDebugMessageCallbackARB`, used to specify the callback function, takes two arguments: the name of the callback function and a pointer to the user data. The pointer to the user data can only be changed with a new call to `glDebugMessageCallbackARB`. This pointer allows the callback to receive user-

```
void CALLBACK DebugLog(GLenum source, GLenum type, GLuint id, GLenum severity, ↩
    GLsizei length, const GLchar *message, GLvoid *userParam)
{
  printf("Type: %s; Source: %s; ID: %d; Severity: %s\n",
    getStringForType(type).c_str(),
    getStringForSource(source).c_str(),id,
    getStringForSeverity(severity).c_str());
  printf("Message: %s\n", message);
}
```

Listing 33.2. Example of a simple callback function.

defined data, in addition to the event data. Listing 33.3 shows a very simple, although useless, example. In that snippet of code, the last two OpenGL function calls should generate events. The callback function will receive a pointer to `myData`, which will hold the value 2 the first time, and 3 the second.

```
int myData;
...
// set myData as the user data
glDebugMessageCallbackARB(DebugLog, &myData);
...
//set the value of the variable myData
myData = 2
// Generate an event due to an invalid parameter to glEnable
glEnable(GL_UNIFORM_BUFFER);
// change the variable myData.
myData = 3;
// From now on events will carry the value 3 in the user param.
// Another event: parameter combination not available in core profile
glPolygonMode(GL_FRONT, GL_LINE);
```

Listing 33.3. Example of the user-data parameter usage facilities.

A more realistic and complete example of usage for the user data is to pass a struct holding pointers to the relevant subsystems of the application, such as the resource manager and the rendering manager.

Once set, the callback can be disabled by passing NULL as the first argument to `glDebugMessageCallbackARB`. From that point onwards, events will be directed to the message log. A final note: any call to an OpenGL or window-system function inside the callback function will have undefined behaviour and may cause the application to crash.

33.4 Sorting Through the Cause of Events

Getting events reported is only a part of the debugging process. The rest of the procedure involves pinpointing the issue's location and then acting upon it. Finding

the cause of the event or the offending lines of code can be a very simple task with the extension *synchronous* mode.

The specification defines two event reporting modes: *synchronous* and *asynchronous*. The former will report the event before the function that caused the event terminates. The latter option allows the driver to report the event at its convenience. The reporting mode can be set by enabling or disabling `GL_DEBUG_OUTPUT_SYNCHRONOUS_ARB`. The default is asynchronous mode.

In synchronous mode, the callback function is issued while the offending OpenGL call is still in the call stack of the application. Hence, the simplest solution to find the code location that caused the event to be generated is to run the application in a debug runtime environment. This allows us to inspect the call stack from inside the IDE by placing a breakpoint at the callback function.

33.4.1 Accessing the Call Stack Outside the IDE

The other solution, harder to deploy but neater, is to implement a function within the application to get the call stack and print it out. While more complex, this solution is far more productive since there is no need to keep stopping the program at each event; it also allows us to filter from the call stack only the calls originated from the application, eliminating calls to the operating system's libraries, thereby providing a cleaner output. This output can be directed to a stream, allowing for maximum flexibility. The callback, when running in test machines, can produce a sort of a minidump file, pinpointing the error location, that later can be sent to the developer team.

The source code accompanying this chapter provides a small library that includes a function for Windows and Linux systems that retrieves the call stack. An example of a call stack is printed by the library as

```
function: setGLParams - line: 1046
function: initGL - line: 1052
function: main - line: 1300
```

33.5 Accessing the Message Log

As mentioned before, if no callback is defined, the events are stored in an internal log. In the extension specification, this log is referenced as the message log; however, it contains all the event's fields. Hence, *event log* would probably be a more descriptive name.

The log acts as a limited-size queue. The limited size causes new events to be discarded when the log is full; hence, the developer must keep cleaning the log to ensure that new events fit in, as any event that occurs often is bound to fill the log very rapidly. Also, being a queue, the log will provide the events in the order they were added. When retrieving an event, the oldest event is reported first.

```
GLint maxMessages, totalMessages, len, maxLen, lens[10];
GLenum source, type, id, severity, severities[10];
// Querying the log
glGetIntegerv(GL_MAX_DEBUG_LOGGED_MESSAGES_ARB, &maxMessages);
printf("Log Capacity: %d\n", maxMessages);

glGetIntegerv(GL_DEBUG_LOGGED_MESSAGES_ARB, &totalMessages);
printf("Number of messages in the log: %d\n", totalMessages);

glGetIntegerv(GL_MAX_DEBUG_MESSAGE_LENGTH_ARB, &maxLen);
printf("Maximum length for messages in the log: %d\n", maxLen);

glGetIntegerv(GL_DEBUG_NEXT_LOGGED_MESSAGE_LENGTH_ARB, &len);
printf("Length of next message in the log: %d\n", len);
char *message = (char *)malloc(sizeof(char) * len);
// Retrieving all data for the first event in the log. Placing NULL
//  in any of the fields is allowed and the field will be ignored.
glGetDebugMessageLogARB(1, len, &source, &type, &id, &severity, NULL, message);
// Retrieving the severity and messages for the first 10 events.
char *messages = (char *)malloc(sizeof(char) * maxLen * 10);
glGetDebugMessageLogARB(10, maxLen * 10, NULL, NULL, NULL, severities, lens, ↩
    messages);
// Clearing the log
glGetIntegerv(GL_DEBUG_LOGGED_MESSAGES_ARB, &totalMessages);
glGetDebugMessageLogARB(totalMessages, 0, NULL, NULL, NULL, NULL, NULL, NULL);
```

Listing 33.4. Querying the log and retrieving messages.

The log's capacities and the length of the first message can be queried with `glGetInteger`. Multiple events can be retrieved with a single call using `glGetDebugMessageLogARB`. Listing 33.4 presents some examples of usage.

When retrieving multiple events, their associated messages are all concatenated in a string, separated by a null terminator. The array `lens` will store their individual lengths. If the maximum length (second parameter) is not sufficient to hold all the messages to retrieve, only those events whose messages do fit in `message` will actually be retrieved.

Each time an event is retrieved, it is removed from the log. Hence, to clear the log, one just needs to retrieve all events (Listing 33.4).

33.6 Adding Custom User Events to the Log

An application, or third-party library, can also take advantage of the debug facilities now available with this extension since it is possible to insert the application's own events into the log. The log can then be used as a centralized debug resource for all that is related to the graphics pipeline.

One note, though, regarding this approach. Developers of libraries have to be aware of possible clashes in the IDs of the events, although this could partially be

```
glDebugMessageInsertARB(GL_DEBUG_SOURCE_APPLICATION_ARB,
  GL_DEBUG_TYPE_ERROR_ARB,
  1111, GL_DEBUG_SEVERITY_LOW_ARB,
  -1, // null terminated string
  "Houston, there's some problem with my app...");
```

Listing 33.5. Extension function to add events to the event log.

solved by adding, in each event's message, an identification of the library who issued it.

Adding events to the log is very straightforward, with only one new function, `glDebugMessageInsertARB`. An example of usage can be found in Listing 33.5. The source field can only be `GL_DEBUG_SOURCE_APPLICATION_ARB` or `GL_DEBUG_SOURCE_THIRD_PARTY_ARB`. The specification states that low-severity events are not enabled by default. Next section will show how to enable and disable classes, or individual events.

33.7 Controlling the Event Output Volume

The extension provides a function, `glDebugMessageControlARB`, which can be used to filter the events that get reported. The function effectively filters out, `GL_FALSE`, or allows the inclusion, `GL_TRUE`, of any events that match the criteria specified. This does not affect events already in the log; it will only filter new events. Listing 33.6 shows some examples of usage.

To specify a particular combination of `source`, `type`, and/or `severity` (the first three parameters), just set them to either one of the defined enumeration values or use `GL_DONT_CARE`, i.e., no filter is applied to that field.

A set of event IDs can also be specified to set an array with the required values. This only works for pairs of `source` and `type`, both not being simultaneously `GL_DONT_CARE`, while `severity` must be set to `GL_DONT_CARE`. This is because an event is uniquely identified by its `type`, `source`, and `ID`.

This feature can become even more powerful if integrated with a dynamic debugging system. In conjunction with the callback function, or with a periodic inspection to the event log, it is possible to devise a mechanism by which some events are disabled after some number of occurrences.

33.8 Preventing Impact on the Final Release

As we begin to use this extension, its functions will start to appear here and there in our code. This brings up the next issue: how to get rid of all these function calls for the final release version. Building a library allows us to concentrate these calls in a

```
// Disabling events related to deprecated behaviour
glDebugMessageControlARB(GL_DONT_CARE,
  GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR_ARB,
  GL_DONT_CARE,
  0, NULL, GL_FALSE);
// Enabling only two particular combinations of source, type and id.
// Note that first we had to disable all events.
GLuint id[2] = {1280, 1282};
glDebugMessageControlARB(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE,
  0, 0, FALSE);
glDebugMessageControlARB(GL_DEBUG_SOURCE_API_ARB,
  GL_DEBUG_TYPE_ERROR_ARB,
  GL_DONT_CARE,
  // 2 is the number of IDs
  2, id, GL_TRUE);
```

Listing 33.6. Filtering events.

particular class, yet this does not solve the issue, as now one has to call the library's functions instead.

A possible workaround for this issue is to use compilation flags, dealt by the preprocessor. A simple example is shown in Listing 33.7. With this approach, all calls related to the extension can be removed just by undefining the compilation flag.

```
#ifdef OPENGL_DEBUG
// do this for all extension functions
#define GLDebugMessageControl(source, type, sev, num, id, enabled) \\
   glDebugMessageControlARB(source, type, sev, num, id, enabled)
#else
// do this for all extension function
#define  GLDebugMessageControl(source, type, sev, num, id, enabled)
#endif
// now instead of calling glDebug... call GLDebug..., for instance
GLDebugMessageControl(NULL, NULL, NULL, -1, NULL, GL_TRUE);
```

Listing 33.7. Using compilation flags to prevent impact on the final release.

33.9 Clash of the Titans: Implementation Strategies

AMD and NVIDIA have started in different directions when implementing this extension. AMD had a head start since it already had an implementation for `AMD_debug_output`. This extension is very similar to `ARB_debug_output` and contains most of the features in it.

AMD drivers, Catalyst 11.11, are focused on giving more meaningful information to situations typically reported by `glError`. They also consider GLSL

compiler/linker issues as events, providing the *info log* as the message. NVIDIA, on the other hand, started off paying little attention to these issues and went on to provide information on operations such as buffer binding and memory allocation. Starting from version 290.xx, NVIDIA also began to provide some more information to `glError` scenarios, starting to close the gap between their drivers and AMD drivers.

Regarding implementation constants, both drivers share the same log queue size, 128 events, and maximum message length, 1024 bytes.

As for events caused by OpenGL commands, consider binding a buffer to a non–buffer-object name. NVIDIA does not report any event on this, while AMD provides the following information:

```
glBindBuffer in a Core context performing invalid operation
with parameter <name> set to '0x5' which was removed
from Core OpenGL (GL_INVALID_OPERATION)
```

Note that, although the message is not entirely correct since it refers incorrectly to a deprecated feature, it provides the value of the offending parameter. Similar information detail is obtained when attempting to use a deprecated combination of parameters. For instance when using `glPolygonMode(GL_FRONT, GL_LINE)`, one gets

```
Using glPolygonMode in a Core context with parameter
<face> and enum '0x404' which was removed from
Core OpenGL (GL_INVALID_ENUM)
```

NVIDIA, on the other hand, reports

```
GL_INVALID_ENUM error generated. Polygon modes for <face> are
 disabled in the current profile.
```

Although not as complete as AMD, it is an improvement from its earlier implementations (285.62 drivers), where it reported only a `GL_INVALID_ENUM` error.

In a different scenario, when the name is actually bound and data are successfully sent to the buffer, the AMD driver keeps silent, while NVIDIA is kind enough to give information about the operation in low-severity messages:

```
Buffer detailed info: Buffer object 3 (bound to
GL_ELEMENT_ARRAY_BUFFER_ARB, usage hint
is GL_ENUM_88e4) will use VIDEO memory as
the source for buffer object operations.
```

This becomes particularly useful when too many buffers have been allocated and they do not fit in video memory anymore. In this situation, when calling `glBufferData`, the driver reports that system heap memory will be used instead.

NVIDIA also warns if something is about to go wrong. For instance, in a situation where memory is running very low, the following report was issued when calling `glDrawElements`:

```
Unknown internal debug message. The NVIDIA
OpenGL driver has encountered an out of memory
error. This application might behave inconsistently and fail.
```

Regarding performance, the implementations may behave very differently. When rendering in debug mode for a scene with small models with a large number of small VAOs, we noticed some performance degradation with NVIDIA, whereas AMD showed no performance difference. However, when testing with a scene containing a single very large VAO, the performance issue with NVIDIA almost vanished. For both drivers, no significant differences were found regarding the synchronicity mode, which suggests that these drivers have not yet implemented, or optimized, the asynchronous mode. When checking the call stacks, considering a large number of events, there was also no evidence that the asynchronous mode has been implemented.

33.10 Further Thoughts on Debugging

While on the theme of OpenGL debugging, there are a few things that could come in handy. For instance, OpenGL has a rich set of functions to query its state regarding buffers, shaders, and other objects, but all these queries operate on numerical names. For anything other than very simple demos, it becomes hard to keep track of all these numbers. Extension `EXT_debug_label` [Lipchak 11a], available on OpenGL ES 1.1, promotes the mapping of text names to objects.

Another interesting OpenGL ES extension is `EXT_debug_marker` [Lipchak 11b]. This extension allows developers to annotate, with text markers, the command stream for both discrete events or groups of commands. Unfortunately, it does not provide queries for the current active markers.

OpenGL needs more robust development tools that allow shaders to be debugged and state to be inspected. The above two extensions are a step in the right direction and will improve the productivity of developers when used in this context.

33.11 Conclusion

The `ARB_debug_output` extension is a highly welcomed addition, as it adds an extra value to the API, making it possible to evaluate its behavior with a more centralized and efficient approach.

Regarding implementations, AMD had a head start, but NVIDIA is catching up. It may be argued that the extension needs some rewriting so that NVIDIA's approach fits more smoothly. The extension was designed for errors and warnings,

not the type of information NVIDIA is providing, since even a low-severity setting doesn't apply when reporting that an operation has completed successfully. Adding a `GL_DEBUG_INFO` setting would probably be enough to deal with this issue. Still, it is undeniable that this information can come in very handy when problems emerge.

As with any other functionality in OpenGL, the API has a lot of potential, and it is up to the developers to fully unleashed it.

Bibliography

[Konttinen 10a] Jaakko Konttinen. "AMD_debug_output Extension Spec." http://www.opengl.org/registry/specs/ARB/debug_output.txt, June 10, 2010.

[Konttinen 10b] Jaakko Konttinen. "AMD_debug_output Extension Spec." http://www.opengl.org/registry/specs/AMD/debug_output.txt, May 7, 2010.

[Lipchak 11a] Benj Lipchak. "EXT_debug_label Extension Spec." http://www.opengl.org/registry/specs/ARB/debug_output.txt, July 22, 2011.

[Lipchak 11b] Benj Lipchak. "EXT_debug_marker Extension Spec." http://www.opengl.org/registry/specs/ARB/debug_output.txt, July 22, 2011.