KDAB

OpenGL SC

Sean Harmer – Managing Director UK

March 2017

The need for safety-critical systems with user-friendly interfaces is on the rise. To respond to this need, the Khronos Group (responsible for OpenGL, etc) has introduced OpenGL SC, a new standard that enables graphics in safety critical applications.

OpenGL SC certifies a subset of OpenGL ES software and accelerated hardware under DO178-B (for aviation cockpits), ISO26262 (for automotive instrument panels), and ISO 61508 (for industrial automation controls). It primarily focuses on features needed to render aviation cockpit displays: digital instrument panels, 2D maps, 3D terrain, and augmented reality displays.

As an embedded graphics programmer you're already familiar with OpenGL ES, but your company or your customers may begin asking for OpenGL SC support in your products. You may be wondering how much time and effort is needed to port your applications over to OpenGL SC. What techniques won't carry over? Are there functions you can no longer use? What are the gotchas?

This whitepaper reviews the basic differences in moving from OpenGL ES to OpenGL SC, to help you quickly determine what's needed in skills and software to make the change. Specifically, we will compare the latest released version of both standards: OpenGL ES version 3.2 and OpenGL SC 2.0. It's important to note that OpenGL SC 2.0 is not backward compatible with OpenGL SC 1.0; as the 2.0 release of the standard is still relatively recent, it may not yet be available from your OpenGL vendor.

OpenGL SC can't support creating shaders on the fly so you'll need to remove any inline shader compilation/ testing/validation code and switch to shaders created at compile-time.

No Runtime Shaders	
Change(s)	• The following functions are removed: `glCompileShader`, `glCreateShader`, `glAttachShader`, `glValidateProgram`, `glShaderBinary`, `glShaderSource`, `glLinkProgram`, `glGetShaderiv`, `glGetShaderInfoLog`, `glGetShaderPrecisionFormat`, `glGetShaderSource`, `glGetAttachedShaders`
Rationale	• Safety critical code doesn't need to create shaders on the fly. This moves all shader compilation to compile-time and removes the need to include complex (and difficult to validate) GLSL compilers in the OpenGL SC drivers
Feature(s) Possibly Impacted	• Code that compiles shaders at run-time for cross-platform compatibility or toolchain simplicity
Workarounds	• Remove any C/C++ shader compilation/ testing/validation code and switch to compile-time shader compilation

No Frame Buffer to Texture Transfers	
Change(s)	• The following functions are removed: `glCopyTexImage2D`, `glCopyTexSubImage2D`
Rationale	• Reading frame buffer into texture is used in games for special visual effects, mirrors, etc. There is no need to read screen pixels into a texture for safety critical apps
Feature(s) Possibly Impacted	• Special post-processing visual filters applied to screen data (for example, contrast enhancement) • Debug routines that grab screen shots for testing/validation purposes
Workarounds	• Use `glReadnPixels` instead

No glDrawElements	
Change(s)	• The `glDrawElements` function is removed. Programs must use `glDrawRangeElements` instead (added in OpenGL ES 3.0)
Rationale	• Ensuring that the indicies passed to the OpenGL function will not exceed a predefined range helps limit the extent of validation testing
Feature(s) Possibly Impacted	• All functions doing rendering
Workarounds	• Create a wrapper for `glDrawRangeElements` that omits start/ end parameters for non-safety critical builds and calls `glDrawElements` instead

No Object Deletion	
Change(s)	• The following functions are removed: `glDeleteBuffers`, `glDeleteFramebuffers`, `glDeleteProgram`, `glDeleteRenderbuffers`, `glDeleteShader`, `glDeleteTextures`
Rationale	• Safety critical programs do not dynamically allocate memory, as memory exhaustion would cause failure. Routines that dynamically release resources are not needed because the program will not stop executing or return to the operating system
Feature(s) Possibly Impacted	• Libraries containing functions using exception-safe or Resource Acquisition Is Initialization (RAII) resource management techniques
Workarounds	• Create a safety-critical branch that omits resource deletion for shared sources • Use `#ifdefs` to wrap or remove resource deletion for safety critical builds

Safety critical programs do not dynamically allocate memory, as memory exhaustion would cause failure. And memory cleanup isn't needed as the program shouldn't stop executing or return to the operating system.

No glCompressedTexImage2D	
Change(s)	• The `glCompressedTexImage2D` function is removed. Programs must use `glCompressedTexSubImage2D` instead
Rationale	• `glCompressedTexImage2D` re-allocates memory for a given texture, and reallocating is forbidden. One must use `glTexStorage2D` to allocate storage (only once), then upload data into the texture via `glTexSubImage`
Feature(s) Possibly Impacted	• Any graphics using compressed textures like background images or dial/gauge texturing
Workarounds	• Create wrapper for `glCompressedTexImage2D` that calls `glCompressedTexSubImage2D` and grabs full image • Use standard compression formats

No Object Verification	
Change(s)	• The following functions are removed: `glIsBuffer`, `glIsFramebuffer`, `glIsRenderbuffer`, `glIsShader`, `glIsTexture`
Rationale	• Typically functions that are used to test for valid objects before freeing resources are not needed if no resources can be freed
Feature(s) Possibly Impacted	• Verification checks to ensure APIs being used correctly • Unit test frameworks
Workarounds	• Remove validation • Use `#ifdefs` to wrap or remove resource deletion for safety critical builds

No Cube Maps	
Change(s)	• The function `glTexImage2D` is removed • The functions `glTexParameteri` and `glBindTexture` will not accept a `GL_TEXTURE_CUBE_MAP` parameter • The following constants are removed: `GL_TEXTURE_CUBE_MAP_NEGATIVE_X` `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y` `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z` `GL_TEXTURE_CUBE_MAP_POSITIVE_X` `GL_TEXTURE_CUBE_MAP_POSITIVE_Y` `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`
Rationale	• Texture cubes are typically used for providing background to virtual reality gaming environments and are not necessary for instrument clusters
Feature(s) Possibly Impacted	• Artificial horizon indicator (or "8-ball") for aircraft/spacecraft • Sky texture above a 2.5D navigation map
Workarounds	• Use individual texture for each cube face • Use single tall texture with attribute-specifying faces; use special logic in fragment shader to grab attribute and translate into texture y-offset

In Open GL SC, APIs that don't pass buffer size or length are replaced with an equivalent "n" routine that does (for example, glReadnPixels instead of glReadPixels).

No glReadPixels	
Change(s)	• The `glReadPixels` function is removed. Programs must use `glReadnPixels` instead (added in OpenGL ES 3.2)
Rationale	• Passing size of the buffer allows checks that prevent buffer overruns
Feature(s) Possibly Impacted	• Special post-processing visual filters applied to screen data (for example, contrast enhancement) • Debug routines that grab screen shots for testing/validation purposes
Workarounds	• If sharing source with code earlier than OpenGL ES 3.2, wrap `glReadPixels` calls with macro that tests against buffer size or discards `bufSize` parameter and calls `glReadnPixels` • Otherwise convert all `glReadPixels` calls to use `glReadnPixels` and add `bufSize` parameter • If using for debugging only, remove calls

No Uniform or Attribute Inspection	
Change(s)	• The following functions are removed without replacements: `glGetActiveAttrib`, `glGetActiveUniform` • The functions `glGetUniformfv`, `glGetUniformiv` and `glGetUniformuiv` are removed. Programs must use `glGetnUniformfv`, `glGetnUniformiv`, or `glGetnUniformuiv` instead (added in OpenGL ES 3.2)
Rationale	• This family of functions are used to inspect the Open GL state. The `glGetActive*` functions are susceptible to buffer overrun unless they use dynamic memory allocation, neither of which is allowable for safety critical applications. The `glGetUniform*` functions are substituted for versions that pass the buffer's size, allowing prevention of buffer overruns
Feature(s) Possibly Impacted	• Debugging functions • Unit tests and test scaffolding • GL state inspection
Workarounds	• If sharing source with code earlier than OpenGL ES 3.2, wrap `glGetUniform*` calls with macro that tests against buffer size or discards `bufSize` parameter and calls `glGetnUniform*` • Convert all `glGetUniform*` calls to use `glGetnUniform*` and add `bufSize` parameter • Remove calls to `glGetActiveAttrib` and `glGetActiveUniform`

The family of functions that inspect the Open GL state are susceptible
to buffer overrun unless they use dynamic memory allocation, neither of which is allowable for safety critical applications.

Conclusion

The future will dramatically increase the number of embedded systems that we consider to be safety-critical and as we become more dependent on those systems, the consequences of their failure become more serious. Hence the need for standards like OpenGL SC.

While implementing a functional safety process may significantly impact your development, it won't be due to OpenGL SC. A greater concern may be making safety-critical code without using memory frees or reallocation – avoiding C++ delete – which may require a bit more restructuring than just a handful of API alterations. The differences between OpenGL ES and OpenGL SC are limited to a handful of functions, many of which aren't in common use. This breakdown shows that adopting OpenGL SC requires minimal changes to your existing OpenGL ES source code and habits, which should be somewhat of a relief.

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages. Founded in 1999, KDAB has offices throughout North America and Europe.

KDAB

www.kdab.com

© 2017 the KDAB Group. KDAB is a registered trademark of the KDAB Group. All other trademarks belong to their respective owners.