Embedded Systems project 2024/2025
OpenGL for Safety Critical Systems

Federico Valentino

1 Introduction

This document briefly describes the differences in features of the various OpenGL implementations, focusing on OpenGL SC.

2 OpenGL ES

OpenGL ES is designed to operate on mobile devices, embedded systems and consoles. It removes features from OpenGL in order to increase performance and power efficiency.

2.1 Feature Comparison

Feature	OpenGL 4.6	OpenGL ES 3.2
Fixed-function pipeline	Yes	No(Only Shaders)
Immediate mode	Yes	No
Tesselation Shaders	Yes	No
Geometry Shaders	Yes	Limited
Compute Shaders	Yes	Yes(Limited)
Direct State Access	Yes	No
Separate Shaders Objects	Yes	No
Quads & Polygons	Yes	No(Only triangles)
Multiple Render Targets	Yes	Yes(Limited)
Cube Map Arrays	Yes	Yes(Limited)
High Precision Fragment Shaders	Yes	Limited
Advanced Blending Modes	Yes	Limited
Texture Border Colors	Yes	No
3D Textures	Yes	Yes
Transform Feedback	Yes	Yes

Table 1: OpenGL vs OpenGL ES

3 OpenGL SC

OpenGL SC is a subset of OpenGL ES, which as we have already seen, is itself a subset of OpenGL. It is designed to meet the standards of the safety critical market for avionics, industrial, military, medical and automotive applications.

3.1 Feature Comparison

The following tables describe the main changes from OpenGL ES 3.2 to OpenGL SC 2.0. The following tables where obtained from the following article by KDAB

No Runtime Shaders	
Changes	The following functions are removed: glCompileShader, glCreateShader, glAttachShader, glValidateProgram, glShaderBinary, glShaderSource, glLinkProgram, glGetShaderiv, glGetShaderInfoLog, glGetShaderPrecisionFormat, glGetShaderSource, glGetAttachedShaders
Why	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 GLSL compilers in the OpenGL SC drivers
WorkArounds	Remove any C/C++ shader compilation/testingvalidation code and switch to compile-time shader compilation

No Frame Buffer to Texture Transfers	
Changes	The following functions are removed: glCopyTexImage2D, glCopyTexSubImage2D
Why	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
WorkArounds	Use glReadnPixels instead

No glDrawElements	
Changes	The glDrawElements function is removed. Programs must use glDrawRangeElements instead (added in OpenGL ES 3.0)
Why	Ensuring that the indicies passed to the OpenGL function will not exceed a predefined range helps limit the extent of validation testing
WorkArounds	Create a wrapper for glDrawRangeElements that omits start/ end parameters for non-safety critical builds and calls glDrawElements instead

No glCompressedTexImage2D	
Changes	The glCompressedTexImage2D function is removed. Programs must use glCompressedTexSubImage2D instead
Why	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 glTex-SubImage
WorkArounds	• Create wrapper for glCompressedTexImage2D that calls glCompressedTexSubImage2D and grabs full image • Use standard compression formats

No Object Verification	
Changes	The following functions are removed: glIsBuffer, glIsFramebuffer, glIsRenderbuffer, glIsShader, glIsTexture
Why	Typically functions that are used to test for valid objects before freeing resources are not needed if no resources can be freed
WorkArounds	• Remove validation • Use #ifdefs to wrap or remove resource deletion for safety critical builds

No glReadPixels	
Changes	The glReadPixels function is removed. Programs must use glReadnPixels instead (added in OpenGL ES 3.2)
Why	Passing size of the buffer allows checks that prevent buffer overruns
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	
Changes	• 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)
Why	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
WorkArounds	• Convert all glGetUniform* calls to use glGetnUniform* and add bufSize parameter • Remove calls to glGetActiveAttrib and glGetActiveUniform

4 Porting of glxgears to OpenGL SC

The porting of an application from OpenGL to OpenGL SC is not straight forward as it could seem. I choose to port glxgears since it was a small and simple OpenGL demonstration program that came with old GNU/Linux distribution.

(a) Original application

(b) Porting in OpenGL SC

4.1 Shaders

Glxgears used a Fixed Function Pipeline to render three different gears rotating together over a black background; this rendering methodology however was completely removed from OpenGL SC as for OpenGL ES it was decided to use only shader rendering. This means that every instruction related to how the shading or vertex transformation are implemented during the rendering has to be moved either to the vertex or fragment shader. One of the problems in implementing shaders, however, is that OpenGL SC only accepts shaders compiled from GLSL version 1.00. This means that not all transformations can be computed on the shaders. For example, the normal matrix, computed by inverting and transposing the model matrix, has to be computed on the CPU since GLSL 1.00 doesn't have any implementation of those functions. The following is the implemented vertex and fragment shader code:

```
#version 100  // OpenGL SC uses GLSL 1.00
attribute vec3 aPos;
attribute vec3 aNorm;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix;
varying vec3 Normal;
varying vec3 FragPos;
void main() {
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   FragPos = vec3(model * vec4(aPos, 1.0));
   Normal = normalize(normalMatrix * aNorm);
}
```

```
#version 100"  // OpenGL SC uses GLSL 1.00
precision mediump float;
varying vec3 Normal;
varying vec3 FragPos;
uniform vec4 color;
void main() {
   vec3 ambient = vec3(0.3, 0.3, 0.3);
   vec3 lightPos = vec3(5.0, 5.0, 10.0);
   vec3 norm = normalize(Normal);
   vec3 lightDir = normalize(lightPos - FragPos);
   float diff = max(dot(norm, lightDir), 0.0);
   vec3 diffuse = diff * vec3(1.0, 1.0, 1.0);
   vec3 result = (ambient + diffuse) * vec3(color);
   gl_FragColor = vec4(result, 1.0);
}
```

The main difference from a modern shader implemented in GLSL version 4.00+ is that we don't have ins or outs for global variables. Instead we need to use the keyword varying. This was changed in modern version of GLSL with the introduction of geometry shaders which are not present in OpenGL SC. The lighting model implemented in the original glxgears wasn't specified but it was simple, therefore I decided to go with a Lambertian model with a single light source. The shaders still allow for the implementation of more complex lighting models, though this will negatively affect performance. Another major change regarding shaders is that they must be precompiled and saved in a binary container identified by a specific binary format. The shaders will then be loaded by the main program allowing images to be rendered on the screen.

4.2 Vertex generation

The original glxgears renders the gears by generating the vertices in six steps: first, it generates the gear's front face, leaving a circular hole in the center; then, it generates the front face of the teeth; these steps are repeated for the back face; finally, it generates all the necessary faces to connect the front and back faces, including the outer faces linking each front tooth to its corresponding back tooth, as well as the inner faces for the gear's central radius. The application then renders the image by using the primitives GL_QUADS and GL_QUAD_STRIP which, at the time, were much more effective for performance since the data transfer required was significantly lower. Nowadays GPUs are optimized for triangle rendering and OpenGL SC has decided to deprecate GL_QUADS and GL_QUAD_STRIP in favor of GL_TRIANGLES and GL_TRIANGLE_STRIP. This means that, during the porting of glxgears to OpenGL SC, I had to change the vertex generation process to use triangles instead of quads, resulting in a higher number of vertices generated to replicate the final rendering. Image 2 shows how the data usage changes with each primitive. This could have been avoided had I chosen indexed rendering instead. Indexed rendering would have allowed me to generate only the required vertices and then iterate over them with an index array. I decided not to go with this implementation since the original program did not use it either. In addition, the vertices have to be placed in a Vertex Buffer Object [VBO], as you would do in a standard OpenGL implementation. Unlike standard OpenGL, in OpenGL SC Vertex Array Objects [VAO] are not created since they are not available. This means that VBOs will be used directly for rendering.

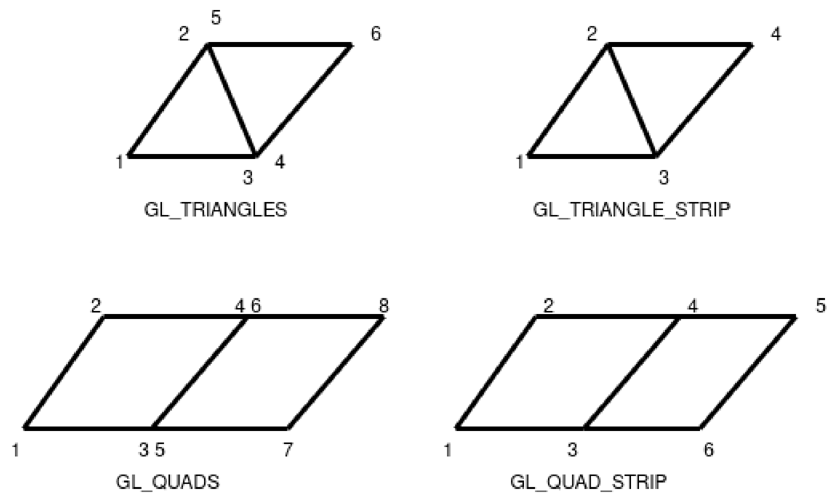

Figure 2: Difference in the various primitives used for rendering

4.3 Draw loop

The draw loop has become more complex since the introduction of shaders requires constant communication between CPU and GPU. As in the original version, the rotation angle of each gear is updated every frame; however in the ported version, the parameters required to draw each gear have to be explicitly passed to the GPU via OpenGL calls, which will set the varying parameters in the shaders. At the end of the loop the VBOs containing the vertex data have to be bound to draw the different parts of the gear.

4.4 Performance comparison

The performance were measured on a system equipped with an Intel Core i5-9600k, an RTX 3060 Ti and 24 GB of RAM. The two applications measured the same 720FPS but the OpenGL SC version, as mentioned in the previous sections, has a greater memory usage since the number of vertices per gear is higher due to the use of GL_TRIANGLES as shown in table 2. It can be lowered by using indexed rendering or by using GL_TRIANGLE_STRIP, which is also the recommendation in today's rendering.

	Original glxgears	OpenGL SC porting
Red gear	528	2412
Blue gear	268	1212
Green gear	268	1212

Table 2: Comparison of vertex data

5 Conclusion

During this work I successfully ported an already existing application to OpenGL SC. The process of porting the application was not as easy as I had expected and I was presented with a couple of challenges. The first issue was the shaders compilation process since they had to be compiled externally and not by the application at runtime, as done by standard OpenGL implementations. Then I had to completely rethink the vertex generation process due to the fact that the only allowed rendering primitive was GL_TRIANGLES. This also showed me one of the downsides of OpenGL SC which is the lack of VAO; each gear is rendered in six steps, each step rendering only one part of the gear at a time. Had I had access to VAO I could have rendered each gear in only one render call resulting in faster CPU times. In conclusion, while the porting process presented some technical challenges, it provided valuable insights into the limitations of OpenGL SC and how to work around them effectively.