Adopting OpenGL ES 3.0

OpenGL ES 3.0 is a superset of the OpenGL ES 2.0 specification, so adopting it in your app is easy. You can continue to use your OpenGL ES 2.0 code while taking advantage of the higher resource limits available to OpenGL ES 3.0 contexts on compatible devices, and add support for OpenGL ES 3.0-specific features where it makes sense for your app's design.

Checklist for Adopting OpenGL ES 3.0

To use OpenGL ES 3.0 in your app:

1. Create an OpenGL ES context (as described in Configuring OpenGL ES Contexts), and specify the API version constant for OpenGL ES 3.0:

   ```
   EAGLContext *context = [[EAGLContext alloc]
   initWithAPI:kEAGLRenderingAPIOpenGLES3];
   ```

   If you plan to make your app available for devices that do not support OpenGL ES 3.0, follow the procedure in Listing 2-1 to fall back to OpenGL ES 2.0 when necessary.
2. Include or import the OpenGL ES 3.0 API headers in source files that use OpenGL ES 3.0 API:

   ```
   #import <OpenGLES/ES3/gl.h>

   #import <OpenGLES/ES3/glext.h>
   ```

3. Update code that uses OpenGL ES 2.0 extensions incorporated into or changed by the OpenGL ES 3.0 specifications, as described in Updating Extension Code below.
4. (Optional.) You can use the same shader programs in both OpenGL ES 2.0 and 3.0. However, if you choose to port shaders to GLSL ES 3.0 to use new features, see the caveats in Adopting OpenGL ES Shading Language version 3.0.
5. Test your app on an OpenGL ES 3.0-compatible device to verify that it behaves correctly.

Updating Extension Code

OpenGL ES 3.0 is a superset of the OpenGL ES 2.0 specification, so apps that use only core OpenGL ES 2.0 features can be used in an OpenGL ES 3.0 context without changes. However, some apps also use OpenGL ES 2.0 extensions. The features provided by these extensions are also available in OpenGL ES 3.0, but using them in an OpenGL ES 3.0 context may require at least minor code changes.

Remove Extension Suffixes

The OpenGL ES 2.0 extensions listed below define APIs that are incorporated into the core OpenGL ES 3.0 specification. To use these features in an OpenGL ES 3.0 context, simply remove the extension suffixes from function and constant names. For example, the name of the `glMapBufferRangeEXT` function becomes `glMapBufferRange`, and the `DEPTH_COMPONENT24_OES` constant (used in the `internalformat` parameter of the `glRenderbufferStorage` function) becomes DEPTH_COMPONENT24.

- OES_depth24
- OES_element_index_uint
- OES_fbo_render_mipmap
- OES_rgb8_rgba8
- OES_texture_half_float_linear
- OES_vertex_array_object
- EXT_blend_minmax
- EXT_draw_instanced
- EXT_instanced_arrays
- EXT_map_buffer_range
- EXT_occlusion_query_boolean
- EXT_texture_storage
- APPLE_sync
- APPLE_texture_max_level

Modify Use of Extension APIs

Some features defined by OpenGL ES 2.0 extensions are in the core OpenGL ES 3.0 specification, but with changes to their API definitions. To use these features in an OpenGL ES 3.0 context, make the changes described below.

Working with Texture Formats

The OES_depth_texture, OES_packed_depth_stencil, OES_texture_float, OES_texture_half_float, EXT_texture_rg, and EXT_sRGB extensions define constants for use in the internalformat and type parameters of the glTexImage family of functions. The functionality defined by these extensions is available in the OpenGL ES 3.0 core API, but with some caveats:

- The glTexImage functions do not support internalformat constants without explicit sizes. Use explicitly sized constants instead:

  ```
  // Replace this OpenGL ES 2.0 code:

  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
  GL_HALF_FLOAT_OES, data);

  // With this OpenGL ES 3.0 code:

  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, width, height, 0, GL_RGBA,
  GL_HALF_FLOAT, data);
  ```

- OpenGL ES 3.0 does not define float or half-float formats for LUMINANCE or LUMINANCE_ALPHA data. Use the corresponding RED or RG formats instead.
- The vector returned by depth and depth/stencil texture samplers no longer repeats the depth value in its first three components in OpenGL ES 3.0. Use only the first (.r) component in shader code that samples such textures.
- The sRGB format is only valid when used for the internalformat parameter in OpenGL ES 3.0. Use GL_RGB or GL_RGBA for the format parameter for sRGB textures.

Alternatively, replace calls to glTexImage functions with calls to the corresponding glTexStorage functions. Texture storage functions are available in as core API in OpenGL ES 3.0, and through the EXT_texture_storage extension in OpenGL ES 1.1 and 2.0. These functions offer an additional benefit: using a glTexStorage function completely specifies an immutable texture object in one call; it performs all consistency checks and memory allocations immediately, guaranteeing that the texture object can never be incomplete due to missing mipmap levels or inconsistent cube map faces.

Mapping Buffer Objects into Client Memory

The OES_mapbuffer extension defines the glMapBuffer function for mapping the entire data storage of a buffer object into client memory. OpenGL ES 3.0 instead defines the glMapBufferRange function, which provides additional functionality: it allows mapping a subset of a buffer object's data storage and includes options for asynchronous mapping. The glMapBufferRange function is also available in OpenGL ES 1.1 and 2.0 contexts through the EXT_map_buffer_range extension.

Discarding Framebuffers

The glInvalidateFramebuffer function in OpenGL ES 3.0 replaces the glDiscardFramebufferEXT function provided by the EXT_discard_framebuffer extension. The parameters and behavior of both functions are identical.

Using Multisampling

OpenGL ES 3.0 incorporates all features of the APPLE_framebuffer_multisample extension, except for the glResolveMultisampleFramebufferAPPLE function. Instead the glBlitFramebuffer function provides this and other other framebuffer copying options. To resolve a multisampling buffer, set the read and draw framebuffers (as in Using Multisampling to Improve Image Quality) and then use glBlitFramebuffer to copy the entire read framebuffer into the entire draw framebuffer:

```
glBlitFramebuffer(0,0,w,h, 0,0,w,h, GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

Continue Using Most Other Extensions in OpenGL ES 3.0

Several key features of iOS device graphics hardware are not part of the core OpenGL ES 3.0 specification, but remain available as OpenGL ES 3.0 extensions. To use these features, continue to check for extension support using the procedures described in Verifying OpenGL ES Capabilities. (See also the *iOS Device Compatibility Reference* to determine which features are available on which devices.)

Most code written for OpenGL ES 2.0 extensions that are also present as OpenGL ES 3.0 extensions will work in an OpenGL ES 3.0 context without changes. However, additional caveats apply to extensions which modify the vertex and fragment shader language—for details, see the next section.

Adopting OpenGL ES Shading Language version 3.0

OpenGL ES 3.0 includes a new version of the OpenGL ES Shading Language (GLSL ES). OpenGL ES 3.0 contexts can use shader programs written in either version 1.0 or version 3.0 of GLSL ES, but version 3.0 shaders (marked with a `#version 300 es` directive in shader source code) are required to access certain new features, such as uniform blocks, 32-bit integers and additional integer operations.

Some language conventions have changed between GLSL ES version 1.0 and 3.0. These changes make shader source code more portable between OpenGL ES 3.0 and desktop OpenGL ES 3.3 or later, but they also require minor changes to existing shader source code when porting to GLSL ES 3.0:

- The `attribute` and `varying` qualifiers are replaced in GLSL ES 3.0 by by the keywords `in` and `out`. In a vertex shader, use the `in` qualifier for vertex attributes and the `out` qualifier for varying outputs. In a fragment shader, use the `in` qualifier for varying inputs.
- GLSL ES 3.0 removes the `gl_FragData` and `gl_FragColor` builtin fragment output variables. Instead, you declare your own fragment output variables with the `out` qualifier.
- Texture sampling functions have been renamed in GLSL ES 3.0—all sampler types use the same texture function name. For example, you can use the new `texture` function with either a `sampler2D` or `samplerCube` parameter (replacing the texture2D and textureCube functions from GLSL ES 1.0).
- The features added to GLSL ES 1.0 by the `EXT_shader_texture_lod`, `EXT_shadow_samplers`, and `OES_standard_derivatives` extensions are part of the core GLSL ES specification. When porting shaders that use these features to GLSL ES 3.0, use the corresponding GLSL ES 3.0 functions.
- The `EXT_shader_framebuffer_fetch` extension works differently. GLSL ES 3.0 removes the `gl_FragData` and `gl_FragColor` builtin fragment output variables in favor of requiring fragment outputs to be declared in the shader. Correspondingly, the `gl_LastFragData` builtin variable is not present in GLSL ES 3.0 fragment shaders. Instead, any fragment output variables you declare with the `inout` qualifier contain previous fragment data when the shader runs. For more details, see Fetch Framebuffer Data for Programmable Blending.

For a complete overview of GLSL ES 3.0, see the *OpenGL ES Shading Language 3.0 Specification*, available from the OpenGL ES API Registry.