OpenGL® ES 3.0
Quick Reference Guide
for
Adreno™ GPUs

Qualcomm Confidential and Proprietary

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

Contents

1 About this Document

This document is a quick reference guide to explain how to utilize OpenGL ES 3.0 APIs within your Android applications running on Snapdragon processors with integrated Adreno 3xx GPUs.

2 OpenGL ES 3.0 Overview

OpenGL ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems - including consoles, phones, appliances and vehicles. It consists of well-defined subsets of desktop OpenGL, creating a flexible and powerful low-level interface between software and graphics acceleration. OpenGL ES includes profiles for floating-point and fixed-point systems and the EGL specification for portably binding to native windowing systems. OpenGL ES 1.X: fixed function hardware offering acceleration, image quality and performance. OpenGL ES 2.X: enables full programmable 3D graphics.

OpenGL ES 3.0 is the newest API introduced by Khronos group. This API introduces mostly features from the core desktop OpenGL 3.3 specifications as well as some additional features that were borrowed from desktop OpenGL 4.x specifications. The new API also adds some new fetures targeting mobile market.

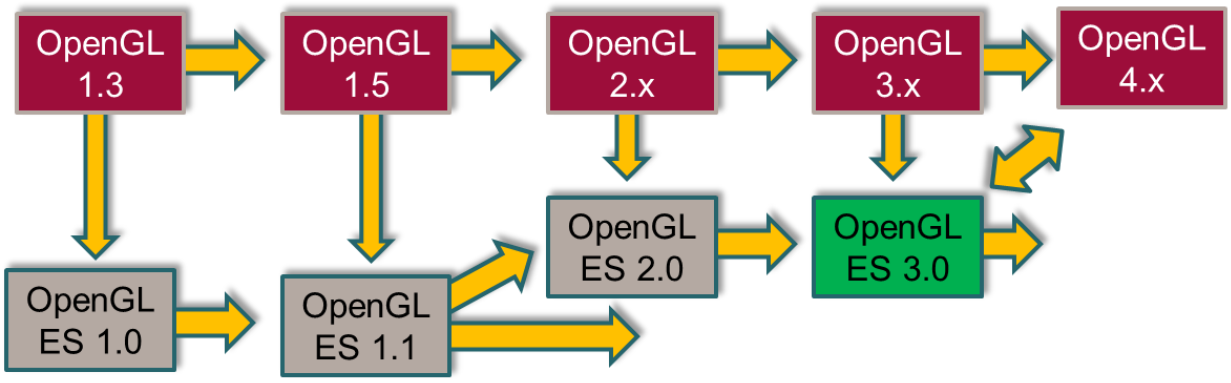


Figure 2-1 OpenGL ES API evolution and relation to desktop OpenGL APIs

3 OpenGL ES 3.0 Features and Usage

3.1 Complete List of OpenGL ES 3.0 Features

Below is the complete list of features that are introduced in this new OpenGL ES 3.0 API specification:

- OpenGL Shading Language ES 3.00
- Transform feedback 1 and 2 (with restrictions)
- Uniform buffer objects including block arrays
- Vertex array objects
- Sampler objects
- Sync objects and fences
- Pixel buffer objects
- Buffer subrange mapping
- Buffer object to buffer object copies
- Boolean occlusion queries, including conservative mode
- Instanced rendering, via shader variable and/or vertex attribute divisor
- Multiple render targets
- 2D array and 3D textures
- Simplified texture storage specification
- R and RG textures
- Texture swizzles
- Seamless cube maps
- Non-power-of-two textures with full wrap mode support and mipmapping
- Texture LOD clamps and mipmap level base offset and max clamp
- At least 32 textures, at least 16 each for fragment and vertex shaders
- 16-bit (with filtering) and 32-bit (without filtering) floating-point textures
- 32-bit, 16-bit, and 8-bit signed and unsigned integer renderbuffers, textures, and vertex attributes
- 8-bit sRGB textures and framebuffers (without mixed RGB/sRGB rendering)
- 11/11/10 floating-point RGB textures
- Shared exponent RGB 9/9/9/5 textures
- 10/10/10/2 unsigned normalized and unnormalized integer textures

- 10/10/10/2 signed and unsigned normalized vertex attributes
- 16-bit floating-point vertex attributes
- 8-bit-per-component signed normalized textures
- ETC2/EAC texture compression formats
- Sized internal texture formats with minimum precision guarantees
- Multisample renderbuffers
- 8-bit unsigned normalized renderbuffers
- Depth textures and shadow comparison
- 24-bit depth renderbuffers and textures
- 24/8 depth/stencil renderbuffers and textures
- 32-bit depth and 32F/8 depth/stencil renderbuffers and textures
- Stretch blits (with restrictions)
- Framebuffer invalidation hints
- Primitive restart with fixed index
- Unsigned integer element indices with at least 24 usable bits
- Draw command allowing specification of range of accessed elements
- Ability to attach any mipmap level to a framebuffer object
- Minimum/maximum blend equations
- Program binaries, including querying binaries from linked GLSL programs
- Mandatory online compiler
- Non-square and transposable uniform matrices
- Additional pixel store state
- Indexed extension string queries

3.2 Key OpenGL ES 3.0 Features and Usage

Below is the detailed description and usage guidelines for some of the key features in OpenGL ES 3.0 API:

3.2.1 sRGB Textures

This feature allows hardware to convert textures to linear color space at the time of texture sampling. Using sRGB textures is an optimization for linear lighting pipelines which result in improved contrast, preserving detail in dark areas of the image.

Usage:

When calling glTexImage2D Specify the *internal format* as GL_SRGB8 or GL_SRGB8_ALPHA8

Notes:

OpenGL ES 3.0 doesn't currently allow for creating sRGB framebuffers so in order to correctly convert from linear space back to gamma space for display a pow(1.0 / 2.2) operation needs to manually be applied to the final image.

3.2.2 ETC2 Texture Compression

ETC2 is a new texture compression format that supports textures with and without alpha channels. ETC2 provides improved quality over ETC1, and frees developers from managing vendor specific compression formats in their pipelines.

Usage:

ETC2 textures can be created using the QCompress tool in the Adreno SDK which is available for download at http://developer.qualcomm.com

3.2.3 Instanced Rendering

Instanced Rendering allows rendering large numbers of objects with a single draw call. Vertex data is shared between objects, but other object properties may vary per instance. Instancing is especially important on mobile GPUs where there is typically a high CPU overhead per draw call.

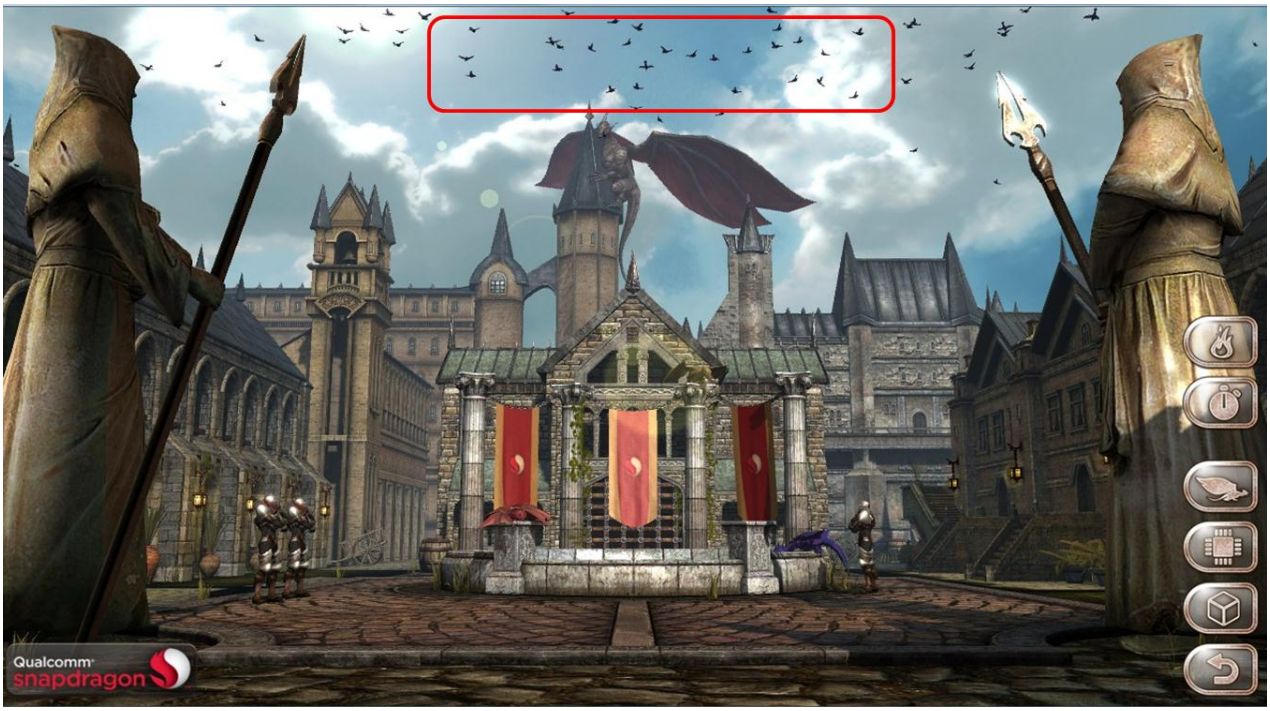


Figure 3-1 Multiple instances of birds are drawn with a single draw call

Primary APIs:

void glDrawArraysInstanced(GLenum mode, GLint first, GLsizei count, GLsizei primcount);
- Draw multiple instances of a range of elements

*void glDrawElementsInstanced(GLenum mode, GLsizei count, GLenum type, const void * indices, GLsizei primcount);*
- Draw multiple instances of a set of elements

void glVertexAttribDivisor(GLuint index, GLuint divisor);
- Modify the rate at which generic vertex attributes advance during instanced rendering

Notes:

Both vertex attribute based instancing, and shader instancing (using gl_InstanceID) are available in OpenGL ES 3.0

3.2.4 Multiple Render Targets

This allows writing to multiple render targets from a fragment shader, reducing or eliminating the need to perform multiple render passes to write data into different targets. MRTs are essential for efficient implementation of deferred shading pipelines.

Sample Usage:

Create FBO with multiple color attachments

```
glGenFramebuffers( 1, &fboHandle);
glBindFramebuffer( GL_FRAMEBUFFER, fboHandle);
```

(create texture 0)

```
glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
textureHandle0, 0);
```

(create texture 1)

```
glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
textureHandle1, 0);
```

(create texture 2)

```
glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D,
textureHandle2, 0);
GLenum buffers[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
GL_COLOR_ATTACHMENT2};
glDrawBuffers(3,buffers);
```

Write to multiple buffers in fragment shader

```
out vec4 gl_FragColor[3];
void main()
{
  gl_FragColor[0] = (color 0…)
  gl_FragColor[1] = (color 1…)
  gl_FragColor[2] = (color 2…)
}
```

For a complete implementation sample please see the non-photo realistic rendering sample in the Adreno SDK available at http://developer.qualcomm.com

3.2.5 PCF (Percentage Closer Filtering) Shadows

PCF samples are a technique for improving the quality of depth mapped shadows by filtering the results of the depth comparison producing softer shadow edges.

Figure 3-2 PCF example for smoother and more realistic dynamic shadows

Usage:

Define the texture sampler as ‘sampler2DShadow’, e.g.

uniform sampler2DShadow shadowMap;

Notes:

To ensure correct results both the min and mag filters for the depth texture bound to the shadow sampler need to be set to ‘GL_LINEAR’

3.2.6 Texture and Renderbuffer Formats

There are numerous new texture and renderbuffer formats that were introduced with OpenGL ES 3.0. These include:

- 16 bit (with filtering) and 32 bit (without filtering) floating point textures
- 11/11/10 floating point RGB textures

- 9/9/9/5 shared exponent RGB textures
- 10/10/10/2 integer textures (unsigned normalized and unnormalized)
- 24 and 32 bit depth renderbuffers and textures
- 28/8 and 32F/8 depth/stencil renderbuffers and textures
- R and RG textures

3.2.7 Uniform Buffers

Use of Uniform Buffers is an optimization for providing uniform data to shader programs. Switching a Uniform Buffer binding is typically faster than switching individual uniform values. In addition Uniform Buffers can store larger blocks of data, and can be shared between different programs.

Sample Usage:

Declare uniform block in shader

```
uniform Transform
{
mat4x4 matModelView;
mat4x4 matProj;
}
```

Create uniform buffer

```
GLuint hUniformBlock = glGetUniformBlockIndex( hShaderProgram, "Transform");
glGetActiveUniformBlockiv( hShaderProgram, hUniformBlock, GL_UNIFORM_BLOCK_DATA_SIZE,
&UniformBlockSize);
glGenBuffers(1, &hTransformBufferName);
glBindBuffer(GL_UNIFORM_BUFFER, hTransformBufferName);
glBufferData(GL_UNIFORM_BUFFER, UniformBlockSize, NULL, GL_DYNAMIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
glBindBufferBase(GL_UNIFORM_BUFFER, 0, hTransformBufferName);
glUniformBlockBinding(hShaderProgram, hUniformBlock, 0);
```

Update the uniform buffer

(update matModelView and matProj….)

```
glBindBuffer(GL_UNIFORM_BUFFER, hTransformBufferName);
```

```
glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(matModelView), &matModelView);
glBufferSubData(GL_UNIFORM_BUFFER, sizeof(matModelView), sizeof(matProj), &matProj);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
(draw…)
```

3.2.8 Transform Feedback

Transform feedback allows for preservation of the post-transform state of primitives processed by a vertex shader to be written out to buffer objects which can be re-submitted for use in subsequent GPU processing. Use of transform feedback can be a substantial optimization for particle systems which otherwise may have had to copy data between the CPU and GPU each frame.

Primary APIs:

*void glTransformFeedbackVaryings(GLuint program, GLsizei count, const char ** varyings, GLenum bufferMode);*

- Specify values to record in transform feedback buffers

*void glGenTransformFeedbacks(GLsizei n, GLuint *ids);*

- Reserve transform feedback object names

void glBindTransformFeedback(GLenum target, GLuint id);

- Bind a transform feedback object

void glBeginTransformFeedback(GLenum primitiveMode);

- Start transform feedback operation

void glEndTransformFeedback(GLenum primitiveMode);

- End transform feedback operation

For a complete implementation sample please see the transform feedback tutorial in the Adreno SDK available at http://developer.qualcomm.com

3.2.9 Vertex Array Objects

Vertex array objects allow for an optimization and/or simplification for binding vertex attributes. After the initialization of the array object the vertex attributes may be bound with a single call. Use of VAOs can allow for driver level optimizations since the vertex layout and relationship between the vertex data, index buffer, etc.. is maintained whenever the VAO is used.

Sample Usage:

Create a VAO and VBO

```
glGenVertexArrays(1, &vaoId);
glBindVertexArray( vaoId );
glGenBuffers(1, &vboId);
glBindBuffer(GL_ARRAY_BUFFER, vboId);
glBufferData(GL_ARRAY_BUFFER, numVerts * 3 * sizeof(GLfloat), verts, GL_STATIC_DRAW);
glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
glBindVertexArray(0);
```

Render using the VAO

```
glBindVertexArray(vaoId])
(draw…)
```

3.2.10 Sampler Objects

Sampler objects provide a mechanism for storing the texture wrap, filter, LOD, and comparison sampling parameters. These parameters may be restored during rendering by simply binding the sampler object

Sample Usage:

Create a sampler object

```
glGenSamplers(1, &samplerObjId);
glSamplerParameteri( samplerObjId, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glSamplerParameteri( samplerObjId, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
glSamplerParameteri( samplerObjId, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glSamplerParameteri( samplerObjId, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Render using the sampler object

```
glBindSampler( textureId, samplerObjId);
(draw…)
```

3.2.11 Framebuffer Invalidate

This provides a hint to the driver that the contents of the specified FBO attachments do not need to be preserved. Use of framebuffer invalidate hints are a significant optimization which prevents the system from wasting valuable cycles copying unneeded data between graphics memory and system memory.

Primary APIs:

*void glInvalidateFramebuffer(GLenum target, GLsizei numAttachments, const GLenum *attachments);*
- Invalidate the contents of attachments within a framebuffer

4 OpenGL ES 3.0 and Android

4.1 OpenGL ES 3.0 and Android 4.3

Google added support for OpenGL ES 3.0 in Android 4.3 and higher (API level 18)

OpenGL ES 3.0 headers and libraries are now available in the Android NDK r9. The NDK r9 contains a sample named 'gles3jni' which demonstrates how to both link directly to the GLESv3 libraries, as well as dynamically acquire GLES3 function pointers using eglGetProcAddress. Applications requiring OpenGL ES 3.0 can set the glEsVersion to "0x00030000" in the application manifest.

4.2 OpenGL ES 3.0 and prior Android versions

Even if Android officially does not expose OpenGL ES 3.0 in java in older versions of Android OS, the OpenGL ES 3.0 API can still be utilized through Android NDK as well as through the Qualcomm provided extension library.

The Qualcomm Adreno SDK v3.2 contains an extension library which simplifies targeting OpenGL ES 3.0 and can be used with versions of Android prior to 4.3. This library exposes GLES3 functions and extensions, it is easy to use. Simple steps of its use include:

- Call Create() method of library object
- Use OpenGL ES 3.0 functions and extensions for example: glDrawArraysInstanced

Below is a sample code snippet:

```
#include "glesextlib.h"
// .. Code to create ES3 context .. eglChooseConfig, eglCreateContext, eglMakeCurrent, etc.
CGLESExtLib extLib;
if (extLib.Create())
{
    // OpenGL ES 3.0 is available
    // Use OpenGL ES 3.0 gl* function calls
}
else
{
    // OpenGL ES 3.0 not available
}
```

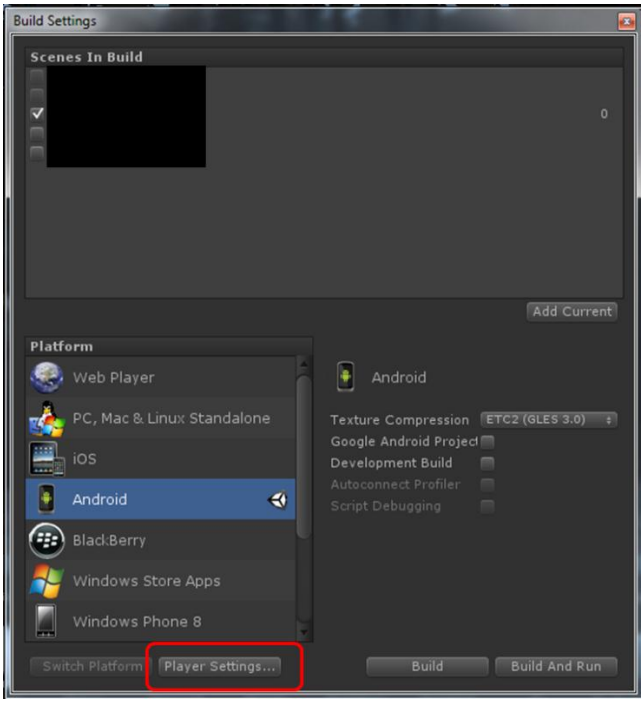
5 OpenGL ES 3.0 and Unity

Unity added support for OpenGL ES 3.0 in Unity 4.2 release. The key features supported by Unity include:

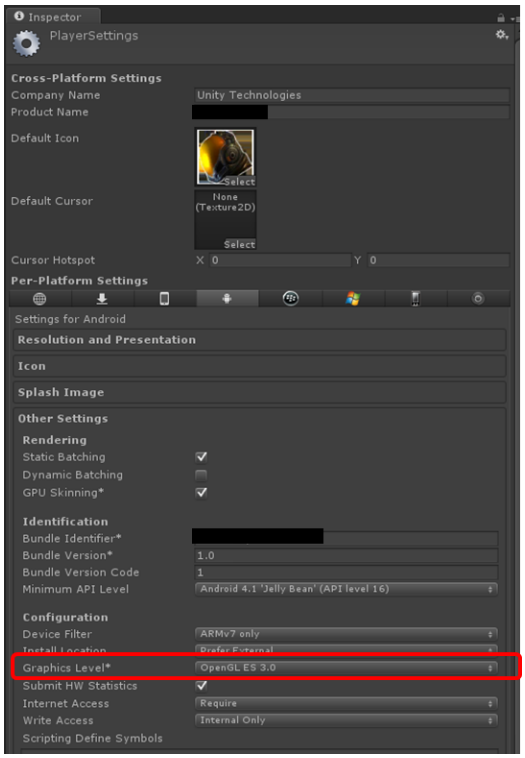
- GPU Skinning using Vertex Transform feedback for more performance and power efficient Character Animation
- Soft Shadows using Percentage Close Filtering for smoother, more realistic dynamic shadows
- ETC2 Texture compression, for new standard cross platform, high fidelity texture compression format
- Deferred Lighting using Multiple Render Targets feature, for more efficient rendering of scenes demanding multiple dynamic lights

Follow simple guidelines as below to enable these OpenGL ES 3.0 features in your Unity application:

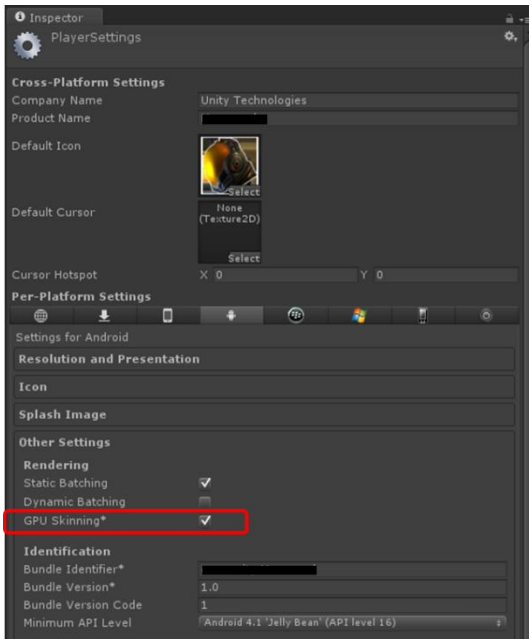
- Step 1: Open File->Build Settings and select “Player Settings…”



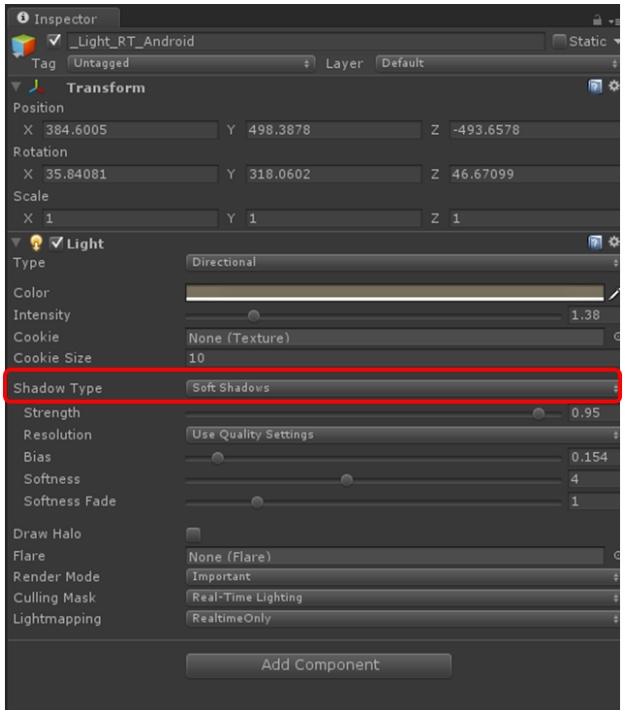
- Step 2: In “Inspector” window, select “Graphics Level” as “OpenGL ES 3.0”. This enables Unity users to select the “OpenGL ES 3.0” features within Unity



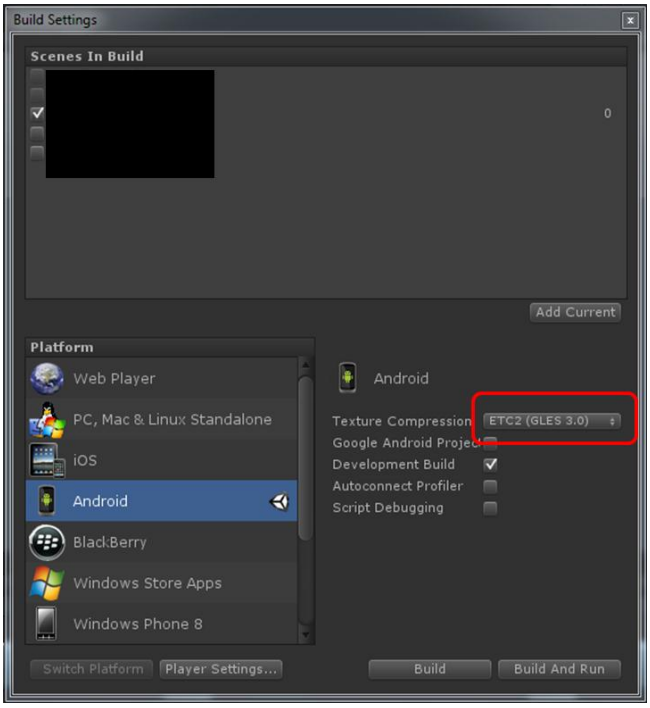
- Step 3: In “Inspector” window, check the box next to “GPU Skinning”. This enables more performance and power efficient character animation using “Transform Feedback”



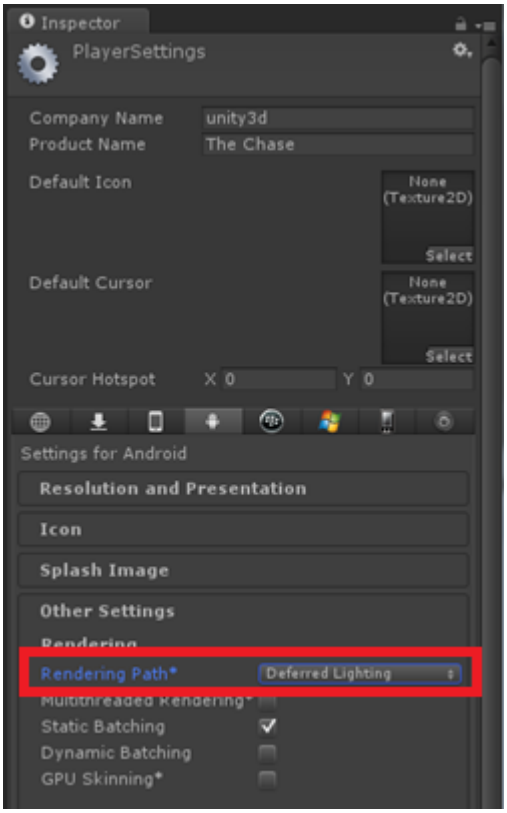
- Step 4: In “Inspector” window for “Light Properties”, select “Shadow Type” as “Soft Shadows”. This enables smoother, more realistic dynamic shadows using “Percentage Close Filtering”



- Step 5: In “Build Settings” window, select “Texture Compression” as “ETC2 (GLES 3.0)”



- Step 6: In “Inspector” window for “Other Settings”, select “Rendering Path” as “Deferred Lighting”. This enables more efficient rendering of scenes where multiple dynamic lights are required.



6 References

Reference documents, which may include Qualcomm standards and resource documents, are listed below. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Ref.	Document
Standards	
S1	*Khronos' "The OpenGL Graphics System: A Specification"*

7 Revision History

Revision	Date	Description
A	August 2013	Initial release
B	February 2014	Update in Unity section to include Deferred Lighting Path