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

39 The ANGLE Project: Implementing OpenGL ES 2.0 on Direct3D

Daniel Koch and Nicolas Capens

39.1 Introduction

The Almost Native Graphics Layer Engine (ANGLE) project is an open-source implementation of OpenGL ES 2.0 for Windows. This chapter explores the challenges that we encountered in the design of ANGLE and the solutions we implemented.

We begin the chapter by providing the motivation for ANGLE and some potential uses of it. We then delve into the implementation details and explore the design challenges that were involved in developing ANGLE. We discuss the feature set that ANGLE provides, including the standard OpenGL ES and EGL extensions that ANGLE supports, as well as some ANGLE-specific extensions [ANGLE 11]. We also describe in detail some of the optimizations that were implemented to ensure high performance and low overhead. We provide performance tips and guidance for developers who may wish to use ANGLE directly in their own projects. We end with some performance comparisons of WebGL implementations using ANGLE and native Desktop OpenGL drivers.

39.2 Background

ANGLE is a conformant implementation of the OpenGL ES 2.0 specification [Khronos 11c] that is hardware-accelerated via Direct3D. ANGLE version 1.0.772 was certified as compliant by passing the ES 2.0.3 conformance tests in October

2011. ANGLE also provides an implementation of the EGL 1.4 specification [Khronos 11b].

TransGaming did the primary development for ANGLE and provides continued maintenance and feature enhancements. The development of ANGLE was sponsored by Google to enable browsers like Google Chrome to run WebGL content on Windows computers that may not have OpenGL drivers [Bridge 10].

ANGLE is used as the default WebGL backend for both Google Chrome and Mozilla Firefox on Windows platforms. Chrome, in fact, uses ANGLE for all graphics rendering, including for the accelerated Canvas2D implementation and for the Native Client sandbox environment.

In addition to providing an OpenGL ES 2.0 implementation for Windows, portions of the ANGLE shader compiler are used as a shader validator and translator by WebGL implementations across multiple platforms. It is used on Mac OS X (Chrome, Firefox, and Safari), Linux (Chrome and Firefox), and in mobile variants of the browsers. Having one shader validator helps to ensure that a consistent set of GLSL ES (ESSL) shaders are accepted across browsers and platforms. The shader translator is also used to translate shaders to other shading languages and to optionally apply shader modifications to work around bugs or quirks in the native graphics drivers. The translator targets Desktop GLSL, Direct3D HLSL, and even ESSL for native OpenGL ES 2.0 platforms.

Because ANGLE provides OpenGL ES 2.0 and EGL 1.4 libraries for Windows, it can be used as a development tool by developers who want to target applications for mobile, embedded, set-top, and Smart TV–based devices. Prototyping and initial development can be done in the developer's familiar Windows-based development environment before final on-device performance tuning. Portability tools such as the GameTree TV SDK [TransGaming 11] can further help to streamline this process by making it possible to run Win32 and OpenGL ES 2.0-based applications directly on set-top boxes. ANGLE also provides developers with an additional option for deploying production versions of their applications to the desktop, either for content that was initially developed on Windows, or for deploying OpenGL ES 2.0–based content from other platforms such as iOS or Android.

39.3 Implementation

ANGLE is implemented in C++ and uses Direct3D 9 [MSDN 11c] for rendering. This API was chosen to allow us to target our implementation at Windows XP, Vista, and 7, as well as providing access to a broad base of graphics hardware. ANGLE requires a minimum of Shader Model (SM) 2 support, but due to the limited capabilities of SM2, the primary target for our implementation is SM3. There are some implementation variances, and in some cases, completely different approaches used, in order to account for the different set of capabilities between SM2 and SM3. Since

SM3 is our primary target, the focus of this chapter is on the description of our implementation for this feature set.

The main challenge of implementing OpenGL ES on top of another graphics API, such as Direct3D, is accounting for different conventions and capabilities. Some differences can be implemented in a straightforward manner, while others are much more involved.

This section begins with one of the most well-known differences between the APIs: the differences in coordinate conventions. Dealing with these in a mathematically sound manner is critical to achieving correct results. Next, we cover another key aspect of the project: the translation of OpenGL ES shaders into their Direct3D equivalents. Following this, we delve into handling data resources such as vertex buffers and textures. Finally, we cover the finer details of the different API paradigms and interfaces, tying the individual aspects into a complete OpenGL ES implementation on top of Direct3D.

39.3.1 Coordinate Systems

It is often said that OpenGL has a right-handed coordinate system and Direct3D has a left-handed coordinate system, and that this has application-wide implications [MSDN 11a]. However, this is not entirely correct. The differences can be best understood by looking at the coordinate transformation equations. Both OpenGL and Direct3D take the position output from the vertex shader in homogeneous (clip) coordinates, perform the same perspective division to obtain normalized device coordinates (NDC), and then perform a very similar viewport transformation to obtain window coordinates. The transformations as performed by OpenGL are shown in Figure 39.1 and Equation (39.1) [Khronos 11c, Section 2.12]. The parameters p_x and p_y represent the viewport width and height, respectively, and (o_x, o_y) is the center

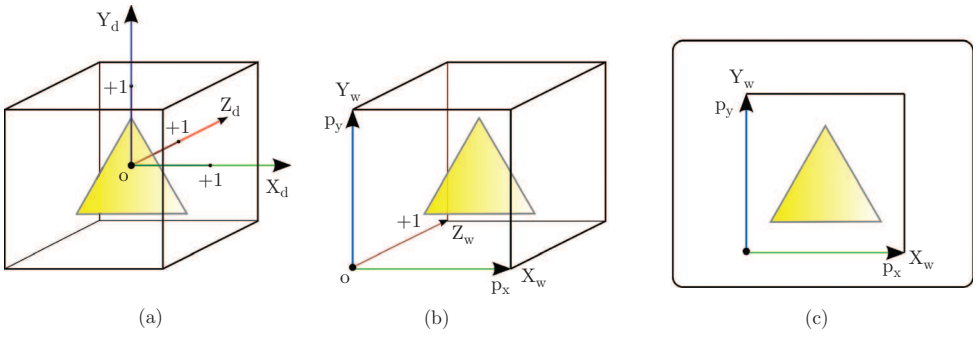


Figure 39.1. GL coordinate spaces: (a) NDC space, (b) window space, and (c) screen space.

of the viewport (all measured in pixels):

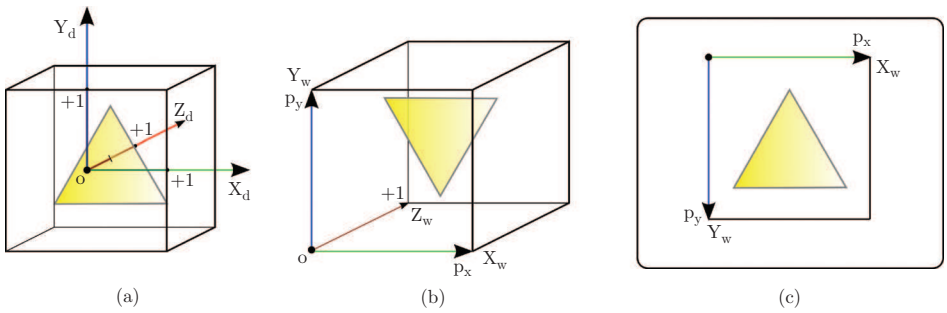
$$\underbrace{\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}}_{\substack{\text{vertex shader} \\ \text{clip coords}}} \rightarrow \underbrace{\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{pmatrix}}_{\substack{\text{perspective division} \\ \text{NDC coords}}} \rightarrow \tag{39.1}$$

$$\underbrace{\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2}x_d + o_x \\ \frac{p_y}{2}y_d + o_y \\ \frac{f-n}{2}z_d + \frac{n+f}{2} \end{pmatrix}}_{\substack{\text{viewport transform} \\ \text{window coords}}} \rightarrow \underbrace{\begin{pmatrix} x_s \\ y_s \end{pmatrix} = \begin{pmatrix} x_w + x_{pos} \\ y_w + y_{pos} \end{pmatrix}}_{\substack{\text{present transform} \\ \text{screen coords}}}.$$

The transformations performed by Direct3D are shown in Figure 39.2 and Equation (39.2) [MSDN 11k]:

$$\underbrace{\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}}_{\substack{\text{vertex shader} \\ \text{clip coordinates}}} \rightarrow \underbrace{\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{pmatrix}}_{\substack{\text{perspective division} \\ \text{NDC coordinates}}} \rightarrow \tag{39.2}$$

$$\underbrace{\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2}x_d + o_x \\ \frac{p_y}{2}(-y_d) + o_y \\ (f-n)z_d + n \end{pmatrix}}_{\substack{\text{viewport transform} \\ \text{window coordinates}}} \rightarrow \underbrace{\begin{pmatrix} x_s \\ y_s \end{pmatrix} = \begin{pmatrix} x_w + x_{pos} \\ p_y - y_w + y_{pos} \end{pmatrix}}_{\substack{\text{present transform} \\ \text{screen coordinates}}}.$$



(a) (b) (c)

Figure 39.2. Direct3D coordinate spaces: (a) NDC space, (b) window space, and (c) screen space.

Window origin. One notable difference between Equations (39.1) and (39.2) is that Direct3D inverts the y-axis during the viewport transformation. Direct3D also considers the window origin to be in the top-left corner with the y-axis pointing down, whereas OpenGL considers the window origin to be in the lower-left corner with the y-axis pointing up, as shown in Figure 39.3. These operations cancel each other out. This means that when the homogeneous coordinates output from an OpenGL vertex shader are fed into Direct3D using the same viewport parameters, the resulting image will correctly appear upright on the screen.

The problem is that when rendering to a texture, the image is stored in window coordinates, and hence, there is no vertical flip performed, leaving the image upside down. Furthermore, the window coordinates are also an input to the pixel shader, so things like fragment coordinates and gradients are inverted.

There are several ways to deal with this issue. The first is to append code to the original OpenGL ES vertex shader to negate the y-component. This way, the negation in the viewport transformation is canceled, meaning the window coordinates are the way OpenGL expects them, and when rendering to a texture, the image appears upright. Since Direct3D flips the image when viewed on screen, we also have to counteract that by explicitly flipping it before the `Present` call. Originally, we chose this option; since negating the y-component in the vertex shader is trivial, it easily solves the render-to-texture issue, and no changes are required to pixel shaders or regular texture handling. It only comes at the cost of an extra pass to copy the rendered result into a texture and flip it upside down before presenting it on the screen. Unfortunately, this pass caused a significant performance impact—up to 20% slower—on low-end hardware when rendering simple scenes.

The second way to deal with the inversion of the y-coordinate is to invert the texture sampling coordinate system by rewriting every sampling operation to use modified texture coordinates: $(s', t') = (s, 1 - t)$. This implies that the data for

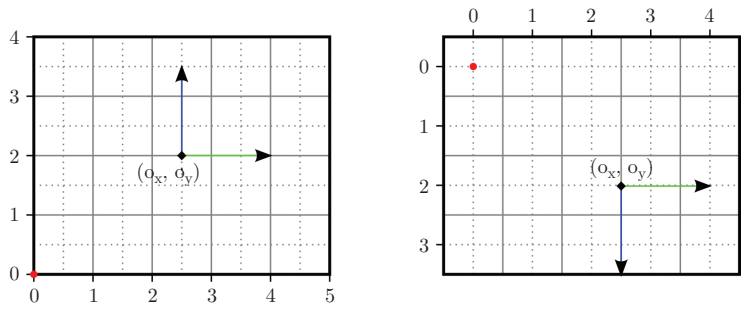


Figure 39.3. Window origin and fragment coordinate differences between OpenGL (left) and Direct3D (right). The red dot represents the location of $(0, 0)$ in window coordinates and (o_x, o_y) is the center of a 5×4 pixel viewport.

regular textures must be stored in an upside down fashion. Cube maps are handled by additionally swapping the top (+Y) and bottom (−Y) faces. It also requires all pixel shader operations that use the window *y*-coordinate to be adjusted. This is the solution currently implemented and is discussed further in Section 39.3.2. The texture inversions on upload are a potential source of inefficiency, but we already swizzle and convert most texture data on load, so this does not add additional overhead. Another concern is that the modification of the texture coordinates turns them into dependent texture reads. This could prevent prefetching of texture data on some GPU architectures, and the extra instructions add computational overhead. Fortunately, this does not appear to be a problem on most desktop GPUs, and we have not observed negative effects due to these modifications.

The third way of solving this issue is to invert the rendering only when rendering into a texture and using the shader unmodified when rendering to a window. This approach could avoid the drawbacks of the first two methods, but it is not without additional implementation complexity. The shaders would have to be compiled differently depending on the rendering destination and it could also affect the fill convention. This approach is still under evaluation and might be implemented in the future.

Winding order. Another interesting consequence of the difference in viewport transformations between OpenGL and Direct3D is that the winding order of a triangle's vertices is reversed. The winding order determines whether a triangle is considered front facing or back facing and hence which primitives are culled. Since the winding order is computed using window coordinates, the need to invert the culling parameters also depends on whether or not the viewport transformation difference is handled in the vertex shader.

Dithering. No specific dithering algorithm is required in OpenGL ES, only that the dithering algorithm depends solely on the fragment's value and window coordinates. When the viewport is inverted, this has the potential to make the dithering algorithm also depend on the viewport height. However, if the identity function is used, this dithering requirement is trivially fulfilled. Direct3D 9 does have a `D3DRS_DITHERENABLE` render state, but dithering is typically no longer directly supported on recent hardware.

Fill convention. One last interesting effect of the different viewport transformations is that it also affects the fill convention. The fill convention is the rule that decides whether a pixel whose center is directly on a triangle's edge is considered covered by that triangle or not. This is vital to prevent adjoining triangles from filling the same pixels twice or leaving gaps. Direct3D enforces a top-left fill convention. OpenGL does not require a specific fill convention, only that a well-defined convention is used consistently. Although ANGLE complies with this, it is worth noting that the OpenGL specification does not guarantee exact pixel results. In particular,

screen-space rectangles should be aligned to pixel edges instead of pixel centers to avoid unexpected results.

Depth range. In addition to the window-origin differences, there is also a difference in the depth range of the homogeneous coordinates that must be accounted for. OpenGL clips the z-coordinate to the $[-1, 1]$ range and then transforms it to the [*near*, *far*] range specified with `glDepthRangef`. Direct3D uses the $[0, 1]$ range instead. Note again that, contrary to popular belief, the z-axis of OpenGL does not point out of the screen. Both OpenGL and Direct3D applications are free to use whichever coordinate system(s) they prefer, as long as the projection takes care of correctly transforming the camera-space coordinates into the intended clip-space coordinates. Since clipping takes place right after the vertex shader stage, we can account for the differences by appending code to the original vertex shader that adjusts the output z-coordinate. We will revisit this in Section 39.3.2.

Fragment coordinates. The numerical coordinates for pixel centers are also different between OpenGL and Direct3D 9. In OpenGL, the pixel centers are located at half-pixel locations, and thus, the (x, y) fragment coordinate of the pixel closest to the origin is $(0.5, 0.5)$. In Direct3D 9, pixel centers are located at integral locations, and the location of the pixel closest to the origin is $(0, 0)$. This also means that viewports are not symmetric around the origin, as shown in Figure 39.3. This oddity has been corrected in Direct3D 10, but for ANGLE on Direct3D 9, a half-pixel offset is required to adjust the fragment coordinates for this difference. This adjustment can also be done at the output of the vertex shader stage, so in homogeneous coordinates, the half-pixel offset becomes $(\frac{1}{p_x} w_c, \frac{1}{p_y} w_c)$.

39.3.2 Shader Compiler and Linker

The initial design of the OpenGL Shading Language was done by 3Dlabs. As part of their work, they developed and released an open-source GLSL compiler front-end and shader validator for the initial version of GLSL [3Dlabs 05]. This GLSL compiler front-end was used as the starting point for ANGLE's shader compiler and translator. The 3Dlabs compiler front-end was designed for Version 1.10 of the GLSL specification and thus needed to be adapted for the GLSL ES Version 1.00 language [Khronos 11e]. The differences between GLSL 1.10 and GLSL ES 1.00 are summed up in a student report from the Norwegian University of Science and Technology [Ek 05].

Architecture. In OpenGL ES 2.0, individual vertex and fragment shaders are compiled using `glCompileShader` and linked into a single program using `glLinkProgram`. With Direct3D 9, however, there is no explicit linking step between the vertex and pixel shaders (the Direct3D equivalent of the ESSL fragment shader). Vertex shader outputs and pixel shader inputs have to be assigned a "semantic" [MSDN 11j], essentially a register identifier, within the HLSL code itself, and they

are implicitly linked when the shaders are made active. Since assigning matching semantics can only be done when both the vertex and pixel shaders are known, the actual HLSL compilation has to be deferred until link time. During the compilation call, ANGLE can only translate the ESSL code into HLSL code, leaving the output and input declarations blank. Note that this is not unique to ANGLE, as other OpenGL and OpenGL ES implementations also defer some of the compilation until link time.

The ANGLE shader compiler component functions as either a translator or a validator. It consists of two main components: the compiler front-end and the compiler back-end. The compiler front-end consists of the preprocessor, lexer, parser, and abstract syntax tree (AST) generator. The lexer and parser are generated from the shading language grammar using the flex [Flex 08] and bison [FSF 11] tools. The compiler back-end consists of several output methods that convert the AST to a desired form of 'object' code. The forms of object code that are currently supported are the HLSL, GLSL, or ESSL shader strings. The shader compiler can validate shaders against either the ESSL specification [Khronos 11e] or the WebGL specification [Khronos 11f]. ANGLE uses the former, and the web browsers use the latter.

During program object linking, the translated HLSL shaders from the "compiled" vertex and fragment shaders are compiled into binary shader *blobs*. The shader blobs include both the Direct3D 9 bytecode for the shaders and the semantic information required to map uniforms to constants. The `D3DXGetShaderConstantTable` method is used to obtain the uniform information and define the mappings between the uniform names and the vertex and pixel shader constant locations. Note that ANGLE uses the Direct3D 10 shader compiler instead of the one included with D3DX9 because it comes as a separately updated DLL, produces superior shader assembly/binary code, and can handle complex shaders more successfully without running out of registers or instruction slots. Unfortunately, there are still some shaders that contain complex conditionals or loops with a high number of iterations that fail to compile even with the Direct3D 10 compiler.

Shader translation. The translation of ESSL into HLSL is achieved by traversing the AST and converting it back into a textual representation while taking the differences between the languages into account. The AST is a tree structure representation of the original source code, so the basic process of turning each node into a string is relatively straightforward. We also extended 3Dlabs' definition of the AST and their traversing framework to preserve additional source information like variable declarations and precisions.

HLSL supports the same binary and unary operators as ESSL, but there are some noteworthy differences in semantics. In ESSL, the first matrix component subscript accesses a column vector while the second subscript (if any) selects the row. With HLSL, this order is reversed. Furthermore, OpenGL constructs matrices from elements specified in column-major order while Direct3D uses row-major order. These differences were addressed by transposing matrix uniforms. This also required

(un-)transposing matrices when used in binary operations. Although it may seem inefficient to transpose matrices within the HLSL shader, at the assembly level it simply results in having multiply-add vector instructions instead of dot-product instructions or vice versa. No noticeable performance impact was observed.

Another significant semantic difference between the two languages is the evaluation of the ternary select operator (`cond ? expr1 : expr2`). With HLSL, both expressions are evaluated, and then the result of one of them is returned based on the condition. ESSL adheres to the C semantics and only evaluates the expression that is selected by the condition. To achieve the ESSL semantics with HLSL, ternary operators are rewritten as if/else statements. Because ternary operators can be nested and statements can contain multiple ternary operators, we implemented a separate AST traverser that hierarchically expands the ternary operators and assigns the results to temporary variables that are then used in the original statement containing the ternary operator. The logical binary Boolean AND (&&) and OR (||) operators also require short-circuited evaluation and can be handled in a similar manner.

To prevent temporary variables and differences in intrinsic function names from colliding with names used in the ESSL source, we "decorate" user-defined ESSL names with an underscore. Reserved names in the ESSL code use the same `gl_` prefix in HLSL, while variables needed to implement or adjust for Direct3D-specific behavior have a `dx_` prefix.

Shader built-ins.

The OpenGL ES shading language provides a number of built-in shader variables, inputs, and functions that are not directly provided by Direct3D's HLSL or that require different semantics between the two languages.

The vertex shading language has no built-in inputs but instead supports application-defined *attribute* variables that provide the values passed into a shader on a per-vertex basis. The attributes are mapped directly to HLSL vertex shader inputs using the `TEXCOORD[#]` semantics.

The *varying* variables form the interface between the vertex and fragment shaders. ESSL has no built-in varying variables and supports only application-defined varyings. These varyings are mapped to HLSL vertex shader outputs and pixel shader inputs using the `COLOR[#]` semantics. In most cases, we could alternatively use the `TEXCOORD[#]` semantics, but these are treated differently for point-sprite rendering, so instead, we always use the `COLOR[#]` semantics for user-defined varyings. The exception for this is in SM2, where variables with the `COLOR[#]` semantics have limited range and precision, and thus we must use the `TEXCOORD[#]` semantic. For this reason, we cannot directly support large points when using SM2.

The vertex shading language has two built-in output variables: `gl_PointSize` and `gl_Position`. The `gl_PointSize` output controls the size at which a point is rasterized. This is equivalent to the HLSL vertex shader PSIZE output semantic, but to meet the GL requirements, it must be clamped to the valid point size range. The `gl_Position` output determines the vertex position in homogeneous coordinates. This is similar to the HLSL vertex shader `POSITION0` output semantic; however, we

```
output.gl_PointSize = clamp(gl_PointSize, 1.0, ALIASED_POINT_SIZE_RANGE_MAX_SM3);
output.gl_Position.x = gl_Position.x - dx_HalfPixelSize.x * gl_Position.w;
output.gl_Position.y = gl_Position.y - dx_HalfPixelSize.y * gl_Position.w;
output.gl_Position.z = (gl_Position.z + gl_Position.w) * 0.5;
output.gl_Position.w = gl_Position.w;
output.gl_FragCoord = gl_Position;
```

Listing 39.1. Vertex shader epilogue.

must account for several differences between Direct3D and OpenGL coordinates at this point before we can use it. As described previously in Section 39.3.1, the x- and y-coordinates are adjusted by the half-pixel offset in screen space to account for the fragment coordinate differences, and the z-coordinate is adjusted to account for the depth-range differences. Listing 39.1 shows the vertex shader epilogue that converts the ESSL shader to Direct3D semantics.

The fragment shading language has three built-in read-only variables: `gl_FragCoord`, `gl_FrontFacing`, and `gl_PointCoord`. The `gl_FragCoord` variable provides the window-relative coordinate values $(x_w, y_w, z_w, 1/w_c)$ for the fragments that are interpolated during rasterization. This is similar to the HLSL VPOS semantic that provides the (x, y)-coordinates. We use the VPOS semantic to provide the base (x, y) screen-space coordinates and then adjust for both the fragment-center and window-origin differences. To compute the z- and w-components of `gl_FragCoord`, we pass the original `gl_Position` from the vertex shader into the pixel shader via a hidden varying. In the pixel shader, the z-value is multiplied by $1/w_c$ to perform perspective correction and is finally corrected by the depth factors calculated from the near and far clipping planes, as shown in Listing 39.2.

The `gl_FrontFacing` variable is a boolean value which is TRUE if the fragment belongs to a front-facing primitive. Under HLSL, similar information is available via the pixel shader VFACE input semantic; however, this is a floating-point value that uses negative values to indicate back-facing primitives and positive values to indicate front-facing ones [MSDN 11h]. This can easily be converted to a boolean value, but we must also account for the different face-winding convention and the fact that point and line primitives are always considered front-facing under OpenGL, whereas the face is undefined for those primitives under Direct3D (see Listing 39.3).

```
rhw = 1.0 / input.gl_FragCoord.w;
gl_FragCoord.x = input.dx_VPos.x + 0.5;
gl_FragCoord.y = dx_Coord.y - input.dx_VPos.y - 0.5;
gl_FragCoord.z = (input.gl_FragCoord.z * rhw)* dx_Depth.x + dx_Depth.y;
gl_FragCoord.w = rhw;
```

Listing 39.2. Calculation of the built-in fragment coordinate for SM3.

```
gl_FrontFacing = dx_PointsOrLines || (dx_FrontCCW ? (input.vFace >= 0.0) : input.↩
    vFace <= 0.0));
```

Listing 39.3. Calculation of front-facing built-in variable.

The `gl_PointCoord` variable provides a set of 2D coordinates that indicate where in a point primitive the current fragment is located. The values must vary from 0 to 1 horizontally (left to right) and from 0 to 1 vertically (top to bottom). These values can be used as texture coordinates in order to provide textured point sprites. Direct3D also has the ability to synthesize texture coordinates for the generated vertices of the point sprite [MSDN 11f]. When this is enabled via the `D3DRS_POINTSPRITEENABLE` render state, the `TEXCOORD` semantic is used to generate texture coordinates that serve as the values for `gl_PointCoord`. Since all points in OpenGL ES are point sprites, we only need to enable this render state once on the Direct3D device initialization.

The OpenGL ES shading language also provides one built-in uniform: `gl_DepthRange`. This is defined as a structure that contains the depth range parameters that were specified via the `glDepthRangef` command in the API. Since HLSL does not provide any built-in uniforms, we pass these parameters in the shaders via a hidden uniform and define and populate the `gl_DepthRangeParameters` structure explicitly in the shader source when referenced by the ESSL code.

Both ESSL and HLSL have a variety of built-in, or intrinsic, functions. Many of the built-in functions have both the same names and functionality, but there are some cases where either the name or functionality is slightly different. Differences in name, such as `frac` (HLSL) and `fract` (ESSL), are easily handled at translation time. In cases where there are functionality differences or simply missing functions, such as `modf` (HLSL) and `mod` (ESSL), this is handled by defining our own functions with the required semantics.

The `OES_standard_derivatives` extension provides the built-in shader functions `dFdx`, `dFdy`, and `fwidth` in the shading language. These gradient computation functions are available in GLSL 1.20 and are commonly used for custom mipmap LOD computations (necessary when using vertex texture fetch) or for extracting screen-space normals. They are translated into the HLSL `ddx`, `ddy`, and `fwidth` intrinsics, respectively, with `ddy` being negated to account for the window origin difference.

The `ANGLE_translated_shader_source` extension [ANGLE 11] provides the ability to query the translated HLSL shader source. This is provided as a debugging aid for developers, as some of the error or warning messages that are reported are relative to the translated source and not to the original shader source.

39.3.3 Vertex and Index Buffers

OpenGL ES 2.0 supports buffer objects that can be used both for vertex and index data, while Direct3D only supports vertex buffers and index buffers separately. This means ANGLE has to wait until a draw call is issued to be able to determine which data can go into which type of Direct3D buffer. Furthermore, Direct3D 9 does not support all of the vertex element types that OpenGL does, so some elements may need to be translated to wider data types. Because this translation can be expensive and not all vertex elements are necessarily used during a draw call, we decided that our baseline implementation should stream the used range of vertex elements into Direct3D vertex buffers sequentially instead of in packed structures. Figure 39.4 shows the basic process.

The streaming buffer implementation uses a Direct3D vertex buffer in a circular manner. New data is appended at the point where the previous write operation ended. This allows the use of a single Direct3D vertex buffer instead of requiring a new one to be created for every draw call. Appending new data is very efficient by making use of the `D3DLOCK_NOOVERWRITE` flag [MSDN 11b] when locking the buffers so that the driver does not need to wait for previous draw calls to complete. When the end of the buffer is reached, the `D3DLOCK_DISCARD` flag is used to allow the driver to rename the buffer. This does not affect data that is already in use by a previous draw call. The streaming vertex buffer only needs to be reallocated when the buffer is not large enough to fit all of the vertex data for a single draw call.

As previously mentioned, unsupported vertex element types need to be translated. Direct3D 9 always supports elements with one to four floating-point values, which offers a universal fallback for any format; however, ANGLE converts the data into more efficient formats whenever possible. For instance, if the Direct3D driver supports the `D3DDECLTYPE_SHORT4N` format, three normalized short values get converted into this format, with the fourth element being set to the default value. We make extensive use of C++ templates in the translation in order to make the code as efficient as possible and avoid writing several dozen customized routines.

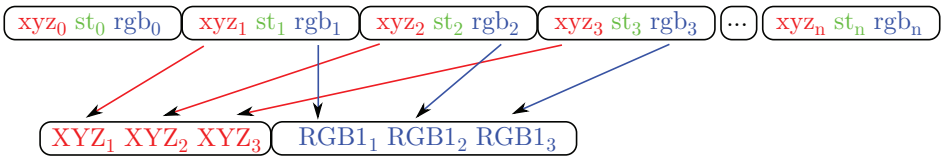


Figure 39.4. Example of streaming buffer translation for a single-triangle draw call. The GL buffer (top) contains position (xyz), texture (st), and color (rgb) vertex data in an interleaved fashion. This is translated into the Direct3D vertex buffer (bottom) that contains the condensed streams of translated position (XYZ) and color (RGB1) values. The texture coordinates and other vertices are not included since they were not referenced by this draw call.

Despite these optimizations, the streaming implementation is not optimal. When drawing the same geometry multiple times, the same data must be resent to the hardware on each draw call. To improve on this, ANGLE also features a static buffer implementation. When the data in a GL buffer is not modified between draw calls, we can reuse the same Direct3D data from the previous draw call that used this buffer. This is achieved by associating a Direct3D vertex buffer with each OpenGL buffer. Instead of streaming data into the global circular vertex buffer, it is streamed into the static buffer on the first use. A cache keeps track of which vertex element formats are stored in the static buffer and at which offset. The static buffer gets invalidated when the OpenGL buffer is subsequently modified or when some formats specified by `glVertexAttribPointer` no longer match those in the cache. ANGLE takes the buffer usage parameter into account when determining whether or not to initially attempt to place data in a static buffer. In our testing, we have also found that some applications set the usage flags incorrectly. Thus, we track whether a nonstatic buffer remains unmodified for a number of uses so it can be heuristically promoted to a static buffer if appropriate. Still, we recommend that applications use the `GL_STATIC_DRAW` hint whenever they are certain a buffer will not be modified after its first use in a draw call.

In addition to vertex array data specified via `glVertexAttribPointer`, OpenGL also supports current attribute values, i.e., attributes that remain constant during a draw call. Direct3D does not have a similar concept. Even though the value stays constant per draw call, using an actual Direct3D vertex shader constant would be complicated because, in one draw call, an attribute can be specified by a vertex array attribute while, in another draw call, it can use the current attribute, and that would require rewriting the Direct3D shader. Instead, we opted to implement current vertex attributes by using vertex buffers with only one element and a stride of zero. When the current attribute value is modified, a whole new Direct3D buffer is created because some drivers do not correctly support updating dynamic buffers that are used with a stride of zero.

ANGLE supports the `OES_element_index_uint` extension, which provides the ability to use 32-bit unsigned integer index buffers with `glDrawElements`. Without this extension, OpenGL ES only supports 8- and 16-bit unsigned indices.

39.3.4 Textures

There are some fundamental texture handling differences between OpenGL and Direct3D. In Direct3D 9, the texture format, usage, and shape (number of mipmaps) must all be declared at texture-object creation time and cannot change over the lifetime of the object. In OpenGL, textures are defined one level at a time and in any level order; the usage is not known in advance, and the shape can change over time as mipmaps are added. Furthermore, any or all levels of a texture can be redefined at any time with the `glTexImage2D` or `glCopyTexImage2D` commands.

In order to handle the differences between Direct3D and OpenGL textures, the application-provided data for each level are stored in a system memory surface. The creation of the Direct3D texture is deferred until draw time when the shape and usage of the texture are known. At Direct3D texture creation time, we must choose whether the texture will be renderable or not. Under OpenGL, any texture can become renderable simply by attaching it to a framebuffer object. Creating all Direct3D textures as render targets can result in degraded performance and can lead to early out-of-memory situations since render target textures are typically pinned to video memory and the driver is unable to page them out to system memory as necessary. Since many textures are never used as render targets, the Direct3D textures are created nonrenderable by default and are loaded with data from the system memory surfaces. This allows the driver to more effectively manage the texture memory. As a consequence of this, whenever a renderable version of a GL texture is required, we create a renderable Direct3D texture and migrate any existing data from either the nonrenderable Direct3D texture or from the system memory surfaces into the renderable texture. We retain the system memory surfaces, which contain the application-provided texture data, as these continue to serve as staging areas for texture updates via `glTexSubImage2D`. The system memory surfaces are also used to avoid reading back texture data from the graphics memory in the cases where the texture is redefined.

Texture redefinition occurs whenever the format or dimensions of level 0 of a texture are changed. When this happens, any existing Direct3D backing texture must be discarded. Ideally, the contents of any existing mip-levels of the texture at that point are preserved in the system memory surfaces. These mip-images should be kept because it is possible for the images to be used again if the texture is later redefined in a way that is consistent with the original data. For example, consider a texture that has four levels with sizes 8×8, 4×4, 2×2, and 1×1. If level 0 is redefined as a 2×2 image, it could be used as a single-level texture with mip-filtering disabled. If level 0 were once again redefined as an 8×8 image with the same format as originally used, this would once again result in a complete texture with four levels, and levels 1 through 3 would have the same data as before. This is the behavior implied by the specification, but not all drivers (including some versions of ANGLE) have correctly implemented image preservation on redefinition. Hence, portable applications should not rely on this behavior; it is recommended that redefining textures be avoided, as this can cause expensive reallocations inside the driver.

OpenGL has recently introduced a new texture creation mechanism that allows the creation of immutable textures: `ARB_texture_storage` [Khronos 11d]. The `glTexStorage` command is used to create a texture with a specific format, size, and number of levels. Once a texture has been defined by `glTexStorage`, the texture cannot be redefined and can only have its data specified by the `gl*SubImage2D` commands or by render-to-texture. This new texture creation API is beneficial for many drivers, as it allows them to allocate the correct amount of memory up front without having to guess how many mip-levels will be provided. ANGLE supports

EXT_texture_storage [ANGLE 11], an OpenGL ES version of this extension, in order to provide a more efficient texture implementation. With the shape and format of the texture known at creation time, we can immediately create a Direct3D texture that corresponds to the GL texture. The system memory surfaces can be omitted because we can load the application-provided texture data directly into the Direct3D9 texture, and we do not have to preserve the system memory copies of the data because TexStorage textures are immutable. The ANGLE_texture_usage extension [ANGLE 11] further provides the ability to let the implementation know the expected usage of a texture. When it is known that a texture will be rendered into, the usage parameter can be specified so that the implementation will know to allocate a renderable texture.

OpenGL also has the notion of *incomplete textures*. This occurs when insufficient levels of a texture are present (based on the filter state), or when the formats or sizes are inconsistent between levels. When sampled in a shader, an incomplete texture always returns the value $(R, G, B, A) = (0, 0, 0, 1)$. Support for incomplete textures is implemented by creating a 1-level 1×1 Direct3D texture of the appropriate type that is bound to the sampler during a draw call.

Another texture difference between OpenGL ES and Direct3D 9 is the set of texture formats that are supported. In addition, OpenGL applications typically supply the texture data in an RGB(A) format, while Direct3D uses a BGR(A) component order for most format types. Because of the difference in component ordering and the limited support for the Direct3D equivalents of some of the packed formats (e.g., the 4444, 5551, and 565 variants), we expand and swizzle texture data into the D3DFMT_A8R8G8B8 format at load time. The main exceptions are the luminance and luminance-alpha unsigned byte texture formats, which are loaded directly as D3DFMT_L8 and D3DFMT_A8L8 textures when natively supported. While we are loading the texture data, we must also flip the texture data vertically to account for the window coordinate differences as described earlier.

To optimize the most common texture loading operations, SSE2 optimized code is used when supported by the CPU. Similarly, glReadPixels requires flipping the image and swizzling the color components, but since it is not expected to be a particularly fast function (because it waits on the GPU to finish rendering), these operations are not yet optimized using SSE2. However, the EXT_read_format_bgra extension is provided in case the application does not require the components to be swizzled.

ANGLE supports a number of extensions that provide a wider range of texture and renderbuffer formats and related capabilities. OES_texture_npot provides support for the full complement of mipmapping, minification filters, and repeat-based wrap modes for nonpower of two textures. 8-bit per component RGB and RGBA renderbuffers (OES_rgb8_rgba8), and BGRA textures (EXT_texture_format_BGRA8888, EXT_read_format_bgra) provide support for 32-bpp rendering as well as exposing formats that do not require conversions for performance reasons. The 16- and 32-bit floating-point texture formats (OES_texture_half_float, OES_

texture_float), including support for linear filtering (OES_texture_half_float_linear, OES_texture_float_linear), are supported in order to provide more precise texture data for algorithms that require it, particularly those that also make use of vertex textures. The DXT1 (EXT_texture_compression_dxt1), DXT3 (ANGLE_texture_compression_dxt3), and DXT5 (ANGLE_texture_compression_dxt5) compressed texture formats [ANGLE 11] are also provided to improve performance by using less texture bandwidth in the GPU and by reducing system and video memory requirements.

39.3.5 Vertex Texture Fetch

OpenGL ES 2 provides the capability to support texture sampling in vertex shaders (also referred to as vertex texture fetch or VTF) although support for this is not mandated. VTF is often used for techniques such as displacement mapping where a heightmap is stored in a texture and then used to adjust the position of the vertex based on the value obtained from a texture lookup. In order to determine whether VTF is supported on a particular implementation and device combination, the application must query the MAX_VERTEX_TEXTURE_IMAGE_UNITS limit. If the value for this limit is zero, VTF is not supported.

The initial implementation of ANGLE did not support vertex textures; however, support was later added as this was a highly sought-after feature. The potential difficulty with implementing VTF in ANGLE is that some SM3 hardware has no support for it, and on other hardware, it is only supported in a very limited form—typically only for 2D textures with 32-bit floating-point formats and only with point filtering. Unlike Direct3D 9, which exposes capabilities like this at a very granular level, OpenGL and OpenGL ES do not provide a way to limit what types of textures or formats can be used with vertex texturing; it is required for all textures types and formats. With OpenGL, using a format or type that is not directly supported by hardware will cause vertex processing to fall back to software. With Direct3D 9, it is possible to get a more complete set of vertex texture capabilities by enabling software vertex processing [MSDN 11g], but that is not without its own drawbacks. First, the Direct3D 9 device must be created with mixed vertex processing and that may not perform as well as a pure hardware device. Next, in order to use a texture with software vertex processing, the texture must be created in the scratch memory pool [MSDN 11b], necessitating additional copies of the texture data and ensuring that they are all kept in sync. Finally, software vertex processing is likely to be significantly slower, and in many cases, developers would rather do without the functionality than have it available but executing in a software fallback.

Unlike SM3 hardware, SM4 (Direct3D 10 capable) hardware does provide full support for vertex textures for all formats, for both 2D and cube textures, and with linear filtering. Furthermore, these capabilities are also exposed via Direct3D 9 on this hardware. As a result, ANGLE only exposes support for vertex texture fetch when it detects that it is running on SM4 hardware and can provide the full com-

plement of vertex texture functionality without falling back to software vertex processing. However, even though SM4 hardware supports 16 vertex texture samplers, the Direct3D 9 API only supports four vertex texture samplers, and thus, this is the maximum supported under ANGLE.

39.3.6 Primitive Types

Both OpenGL and Direct3D provide a number of different types of rendering primitives. They both include primitives for rendering points; line strips and lists; and triangle strips, fans, and lists. OpenGL ES also provides an additional primitive type that is not available under Direct3D 9: the line loop. Line loops are similar to line strips with the addition of a closing line segment that is drawn between the last vertex v_n and the first vertex v_0. Thus, for a render call with n vertices, there are $n - 1$ line segments drawn between vertices $(v_{i-1}, v_i) | 1 \leq i \leq n$ and a final line segment between vertices (v_n, v_0). In ANGLE, this is implemented by drawing two line strips. The first draw call, either arrayed or indexed as specified by the original drawing command, renders the first $n - 1$ line segments via a line strip. The second draw call renders the final line segment using a streaming index buffer that contains the indices of the last and first vertex from the original draw command.

Large points and wide lines are optional capabilities in OpenGL ES 2. Support for these must be queried by checking the maximum available point size range and line width range. Large points are often used for particle systems or other sprite-based rendering techniques, as they have a significant memory and bandwidth saving compared to the fallback method of drawing two triangles forming a screen-aligned quadrilateral. ANGLE supports points with sizes up to a maximum of 64 pixels in order to support point sprite rendering. Wide lines are used less frequently and, as of the time of writing, ANGLE does not support lines with width larger than one.

39.3.7 Masked Clears

Another OpenGL capability not directly supported by Direct3D is masked clear operations. Under Direct3D 9, the color, depth, and stencil masks do not apply to clear operations, whereas they do in OpenGL. Thus, in cases where only some of the color or stencil components are to be cleared, we implement the clear operation by drawing a quad the size of the framebuffer. As with `glClear`, the scissor test limits the area that is affected by the draw command. One of the drawbacks to implementing the clear call via a draw operation is that the current state of the Direct3D device must be modified. Since we do significant caching in order to minimize the state setup that must be done at draw time, this draw command has the potential to interfere with that caching. In order to minimize the individual state changes that must be done, we preserve the current Direct3D rendering state in a stateblock, configure the state for the clearing draw call, perform the draw, and then restore the previous state from the stateblock. In cases where masked clear operations are not required, we directly use the Direct3D 9 `Clear` call for performance.

39.3.8 Separate Depth and Stencil Buffers

Framebuffer configuration in OpenGL ES allows applications to separately specify depth and stencil buffers. Depth and stencil buffers are not separable in Direct3D, and thus, we are not able to support arbitrary mixing of depth and stencil buffers. However, this is not uncommon for OpenGL or other OpenGL ES implementations and can be disallowed by reporting `GL_FRAMEBUFFER_UNSUPPORTED` when separate buffers are simultaneously bound to the depth and stencil binding points. In order to provide support for simultaneous depth and stencil operation, the `OES_packed_depth_stencil` extension is supported in ANGLE. This extension provides a combined depth and stencil surface internal format (`DEPTH24_STENCIL8_OES`) that can be used for renderbuffer storage. In order to use simultaneous depth and stencil operations, the application must attach the same packed depth-stencil surface to both the depth and stencil attachment points of the framebuffer object. The packed depth-stencil format is also used internally for all formats that require only depth or stencil components, and the Direct3D pipeline is configured so that the unused depth or stencil components have no effect. Note that since ANGLE does not yet support depth textures, packed depth-stencil textures are also not supported.

39.3.9 Synchronization

The `glFlush` command is required to flush the GL command stream and cause it to finish execution in finite time. The flush command is implemented in ANGLE via Direct3D 9 event queries [MSDN 11i]. In Direct3D 9, issuing an event query and calling `GetData` with the `D3DGETDATA_FLUSH` parameter causes the command buffer to be flushed in the driver, resulting in the desired effect.

The `glFinish` command is required to block until all previous GL commands have completed. This can also be implemented using Direct3D event queries by issuing an event query and then polling until the query result is available.

ANGLE also supports the `NV_fence` extension in order to provide finer-grained synchronization than is possible with only flush and finish. Fence objects are also implemented via Direct3D 9 event queries as they have very similar semantics.

39.3.10 Multisampling

ANGLE does not currently expose any EGL multisample configurations. This is not due to any inherent technical difficulty, but rather due to lack of demand for it. Support for multisampling is provided with multisampled renderbuffers. The `ANGLE_framebuffer_multisample` extension [ANGLE 11] is a subset of the `EXT_framebuffer_multisample` extension from OpenGL. It provides a mechanism to attach multisampled images to framebuffer objects and resolve the multisampled framebuffer object into a single-sampled framebuffer. The resolve destination can either be another application-created framebuffer object or the window-system provided one.

ANGLE also provides support for copying directly from one framebuffer to another. The `ANGLE_framebuffer_blit` extension [ANGLE 11] is a subset of the `EXT_framebuffer_blit` extension from OpenGL. It adds support for separate draw- and read-framebuffer attachment points and makes it possible to copy directly between images attached to framebuffer objects. `glBlitFramebufferANGLE` is implemented via the Direct3D 9 `StretchRect` function and therefore has some further restrictions compared to the desktop version. In particular, color conversions, resizing, flipping, and filtering are not supported, and only whole depth and stencil buffers can be copied. `glBlitFramebufferANGLE` is also used to resolve multisample framebuffers.

39.3.11 Multiple Contexts and Resource Sharing

ANGLE supports multiple OpenGL ES contexts as well as sharing objects between contexts as described in Appendix C of the OpenGL ES 2.0.25 specification [Khronos 11c]. The object types that can be shared are the resource-type objects: shader objects, program objects, vertex buffer objects, texture objects, and renderbuffer objects. Framebuffer objects and fences are not shareable objects. The requirement to share framebuffer objects was removed from the OpenGL ES 2.0.25 specification in order to be more compatible with OpenGL. In general, it is not desirable to share container-type objects, as this makes change propagation and deletion behavior of the shared objects difficult to specify and tricky to implement and use correctly. Furthermore, there is little value to be had from sharing container objects since they are typically quite small and have no data associated with them.

Shared contexts are specified at context creation time via the `share_context` parameter to `eglCreateContext`. As defined in the EGL 1.4 specification, a newly created context will share all shareable objects with the specified `share_context` and, by extension, with any other contexts with which `share_context` already shares. To implement these semantics, we have a resource manager class that is responsible for creating, tracking, and deleting all shared objects. All nonshared objects, framebuffers and fences, are always managed directly by the context. The resource manager can be shared between contexts. When a new, nonshared context is created, a new resource manager is instantiated. When a shared context is created, it acquires the resource manager from the shared context. Since contexts can be destroyed in any order, the resource manager is reference counted and not directly tied to any specific context.

Direct3D 9 does not have the concept of share groups like OpenGL. It is possible to share individual resources between Direct3D 9Ex devices, but this functionality is only supported on Windows Vista and later. Thus, in order to share resources between ANGLE's GL contexts, the ES and EGL implementations only make use of a single Direct3D 9 device object. The device is created by and associated with the EGL default display and made accessible to each of the GL contexts as

necessary. In order to provide the required separation of state between GL contexts, we must completely transition the Direct3D state when we switch GL contexts. The `eglMakeCurrent` call provides us with the opportunity to do this when the current context is changed. With our state-caching mechanism, we simply need to mark all our cached state dirty at this point, and the necessary Direct3D state will be set for the next draw command.

The current GL context and corresponding EGL display are tracked using thread-local storage (TLS). The TLS is used to hold a pointer to the GL context that has last been made current on this thread via `eglMakeCurrent`. When a GL function call is made, the current GL context for the thread is obtained from the TLS, and the command is dispatched to the GL context. If no GL context is presently current on the thread, the GL command is silently ignored.

ANGLE supports creation of both window- and pbuffer–based EGL surfaces. Window surfaces are implemented by creating a windowed Direct3D 9 swapchain for the EGL surface using the `HWND` window handle that is passed in to `eglCreateWindowSurface` as the native window. The `eglSwapBuffers` command maps to the swapchain's `Present` method. Window resizing is handled by recreating the swapchain. Resizing can be detected either by registering a window handler for the `WM_SIZE` message or by checking the window size at the swap buffer's call. The preferred method is via the window handler, but this does not work for windows that were created in a different process. Pbuffer surfaces, which are used purely for off-screen rendering and do not need to support swapping or resizing, are implemented using Direct3D 9 render-target textures. The `eglBindTexImage` API can also be used to bind a pbuffer as a texture in order to access the contents of the pbuffer.

ANGLE also supports several EGL extensions that enable more efficient integration with applications that use Direct3D directly, such as a browser that uses it internally for the compositor or video decoding. These extensions provide a mechanism that allows textures to be shared between ANGLE's Direct3D device and other Direct3D devices. This also provides the ability to share images between processes since Direct3D resources can be shared across processes. The Direct3D 9 render-target textures that back the pbuffer surfaces can either be created from, or provide, a sharing handle [MSDN 11d]. In order to make use of this, we need a mechanism to provide or extract the Direct3D share handle via EGL.

The `ANGLE_surface_d3d_texture_2d_share_handle` extension [ANGLE 11] allows an application to obtain the Direct3D share handle from an EGL surface. This handle can then be used in another device to create a shared texture that can be used to display the contents of the pbuffer. Similarly, the `ANGLE_d3d_share_handle_client_buffer` extension [ANGLE 11] creates a pbuffer from a Direct3D share handle that is specified via `eglCreatePbufferFromClientBuffer`. This provides the ability to have ES2 content rendered into a texture that has been created by a different Direct3D device. When sharing a surface between Direct3D devices in different processes, it is necessary to use event queries to ensure that rendering to the surface has completed before using the shared resource in another device. From

the ANGLE side, this can be achieved with appropriate use of a fence or by calling `glFinish` to ensure that the desired operations have finished.

39.3.12 Context Loss

Direct3D 9 devices can, under various scenarios, become "lost" [MSDN 11e]. On Windows XP, this can happen when the system has a power management event such as entering sleep mode, or screen saver activation. On Window Vista and later, when using Direct3D 9Ex, device loss is much more infrequent but can still happen if the hardware hangs or when the driver is stopped [MSDN 11d]. When the device is lost, resources that were located in graphics memory are lost, and rendering related operations are ignored. To recover from a lost Direct3D device, the application must release the video memory resources and reset the device.

Unextended OpenGL ES does not provide a mechanism to notify the application of a lost or reset device. EGL does have the `EGL_CONTEXT_LOST` error code that corresponds to the loss of a hardware device. By default, when a device loss occurs, ANGLE generates an out-of-memory error on GL calls, and the context-lost error on EGL calls, to indicate that the context is in an undefined state. In both cases, the proper response is to destroy all the GL contexts, recreate the contexts, and then restore any state and objects as necessary. EGL surfaces do not need to be recreated, but their content is undefined. For details, see Section 2.6 of the EGL 1.4 specification [Khronos 11b]. WebGL applications should additionally follow the advice for Handling Context Lost [Khronos 11a].

ANGLE supports the `EXT_robustness` extension [ANGLE 11], which is based on the OpenGL `ARB_robustness` extension [Khronos 11d], in order to provide a better mechanism for reporting reset notifications. This extension provides an inexpensive query, `glGetGraphicsResetStatusEXT`, which applications can use to learn about context resets. After receiving a reset notification, the application should continue to query the reset status until `GL_NO_ERROR` is returned, at which point the contexts should be destroyed and recreated.

Applications must opt into receiving reset notifications at context creation time by specifying the reset notification strategy attribute as defined in the `EXT_create_context_robustness` extension [ANGLE 11]. Note that even if an application does not opt into receiving reset notifications, or explicitly requests no reset notifications, context loss and resets can still happen at any time. Applications should be made capable of detecting and recovering from these events.

39.3.13 Resource Limits

The OpenGL ES 2.0 API is quite feature-rich; however, there are still some features that are optional or allow for wide variability between implementations. These include the number of vertex attributes, varying vectors, vertex uniform vectors, fragment uniform vectors, vertex texture image units, fragment texture image units,

maximum texture size, maximum renderbuffer size, point size range, and line width range. Applications that need more than the minimum values for any of these limits should always query the capabilities of the GL device and scale their usage based on the device's feature set. Failing to do so and assuming sufficient limits typically results in reduced portability. This is particularly important to keep in mind for WebGL development or when otherwise using a OpenGL ES 2.0 implementation, such as ANGLE, that is provided on desktop hardware. In many cases, the capabilities provided far exceed those available on mobile platforms. The complete listing of minimum requirements for the various implementation-dependent values can be obtained from Tables 6.18–6.20 of the OpenGL ES 2.0.25 specification [Khronos 11c].

Most of the ANGLE limits have been chosen to provide a maximum set of consistent capabilities across a wide range of common hardware. In some cases, the limits are constrained by the Direct3D 9 API even if the hardware has greater capabilities that could be exposed under a different API such as OpenGL or Direct3D 10. In other cases, the limits vary based on the underlying hardware capabilities which are denoted with an asterisk in Table 39.1.

The maximum vertex (254) and fragment (221) uniforms are based on the common capabilities for SM3 vertex (256) and pixel shader (224) constants, but must be lowered to account for the hidden uniforms we may use to implement some of the shader built-ins. The maximum number of varying vectors (10) is the maximum available on SM3 hardware and does not need to be reduced to account for built-in varyings since these are explicitly included in the ESSL varying packing algorithm, as described in Issue 10.16 in the ESSL specification [Khronos 11e]. The maximum texture, cube map, and renderbuffer sizes are directly based on the capabilities of the underlying device, so they range between 2048 and 16384, depending on the hardware.

Capability	ES 2.0 Minimum	ANGLE
`MAX_VERTEX_ATTRIBS`	8	16
`MAX_VERTEX_UNIFORM_VECTORS`	128	254
`MAX_VERTEX_TEXTURE_IMAGE_UNITS`	0	0, 4*
`MAX_VARYING_VECTORS`	8	10
`MAX_FRAGMENT_UNIFORM_VECTORS`	16	221
`MAX_TEXTURE_IMAGE_UNITS`	8	16
`MAX_TEXTURE_SIZE`	64	2048-16384*
`MAX_CUBE_MAP_SIZE`	16	2048-16384*
`MAX_RENDERBUFFER_SIZE`	1	2048-16384*
`ALIASED_POINT_SIZE_RANGE (min, max)`	(1, 1)	(1, 64)
`ALIASED_LINE_WIDTH_RANGE (min, max)`	(1, 1)	(1, 1)

Table 39.1. Resource limits. The most commonly used implementation-dependent values showing both the minimum OpenGL ES 2.0 values and the ANGLE-specific limits. The ANGLE limits are for SM3-capable hardware as of ANGLE revision 889.

39.3.14 Optimizations

To ensure that ANGLE performs as closely as possible to a native implementation of OpenGL ES 2.0, we strive to avoid redundant or unnecessary work, both on the CPU side and on the GPU side.

Much of the effective rendering state is only known at the time of a draw call, so ANGLE defers making any Direct3D render-state changes until draw time. For example, Direct3D only supports explicitly setting culling for clockwise or counterclockwise vertex-winding orders while OpenGL indicates which winding order is considered front-facing by using `glFrontFace`. `glCullFace` determines which of these sides should be culled, and `GL_CULL_FACE` enables or disables the actual culling. In theory, changing any of the `glFrontFace`, `glCullFace`, or `GL_CULL_FACE` states would alter the corresponding Direct3D render state, but by deferring this to the draw call, we reduce it to at most one change (per state) per draw call. For each related group of states, ANGLE keeps a "dirty" flag to determine whether the affected Direct3D states should be updated.

OpenGL identifies resources by integer numbers (or "names"), while the implementation requires pointers to the actual objects. This means ANGLE contains several map containers that hold the associations between resource names and object pointers. Since many object lookups are required per frame, this can cause a noticeable CPU hotspot. Fortunately the associations do not typically change very often, and for currently bound objects, the same name would be looked up many times in a row. Thus, for the currently bound objects like programs and framebuffers, the pointer is cached and replaced or invalidated only when an action is performed which modifies the association.

ANGLE also keeps track of the textures, buffers, and shaders that are currently set on the Direct3D device. To avoid issues with cases where an object gets deleted and a new one coincidentally gets created at the same memory location, resources are identified by a unique serial number instead of their pointer.

Another place caching plays a critical role in optimizing performance is in applying the vertex attribute bindings. Direct3D 9 requires all attributes to be described in a vertex declaration. Creating and later disposing of this object takes up valuable time and potentially prevents the graphics driver from minimizing internal state changes, so a cache was implemented to store the most recently used vertex declarations.

We also endeavor to minimize the GPU workload, both in terms of data transfers to/from the GPU and in terms of computational workload. As discussed earlier, we have eliminated the overhead in flipping the rendered image at presentation time, added support for buffers with static usage, implemented mechanisms to minimize texture reallocations, and used direct clear operations when masked clears are not required. We also optimize out the computation of any shader built-in variables that are not used in the shaders.

It is important to note that while all these optimizations have made ANGLE more complex, they have also significantly helped ensure that the underlying

hardware, accessed through Direct3D, is used as efficiently as possible. Native driver implementations of OpenGL and OpenGL ES also require many of the same optimizations and inherent complexity in order to achieve high performance in practice.

39.3.15 Recommended Practices

Throughout this chapter, we have touched on a variety of practices that should help improve the performance and portability of applications. While these recommendations are targeted specifically at ANGLE's implementation, we expect that many of these practices will also be applicable to other GL implementations:

- Always check for optional features and validate resource limits.
- Group objects in buffers based on data format (type and layout) and update frequency.
- Ensure that appropriate buffer usage flags are used.
- Use static buffers and fully specify the contents of buffers before draw time.
- Use separate buffers for index and vertex data.
- Use immutable textures when available. If `EXT_texture_storage` is not supported, ensure that a complete texture is created and consistently defined.
- Avoid redefining the format or size of existing textures, and create a new texture instead.
- Use the `BGRA_EXT` / `UNSIGNED_BYTE` texture format to minimize texture conversions on load and for pixel readback.
- Use packed depth-stencil for combined depth and stencil support.
- Opt in to reset notifications, and handle context resets appropriately.
- Avoid masked clear operations.
- Avoid line loops by drawing closed line strips instead.
- Use fences instead of `glFinish` for finer synchronization control.
- Avoid using complex conditional statements and loops with a high maximum number of iterations in shaders.

39.3.16 Performance Results

At the time of this writing, there are no de facto benchmarks for WebGL. To correctly interpret performance results of applications and demos, one should first realize that once a draw call command reaches the GPU driver, there are, in theory, few fundamental differences between OpenGL and Direct3D. For ANGLE, in particular, the ESSL and HLSL shaders are largely equivalent, so the GPU performs essentially the same operations. Therefore applications or demos with high numbers of vertices or high levels of overdraw do not really test the graphics API implementation but rather the hardware performance itself.

Potential differences in performance between ANGLE and native OpenGL implementations would stem mainly from the graphics commands issued between draw calls (texture, buffer, and uniform updates), the setup work performed to translate a GL draw call into a Direct3D draw call, and the vertex-shader epilogue and pixel-shader prologues. Therefore, the applications and demos we chose to use for performance comparisons perform a relatively high number of draw calls, use various textures, and/or use nontrivial animations.

The results, shown in Table 39.2, reveal that ANGLE typically performs on par with desktop OpenGL drivers. This demonstrates that on Windows, it is viable to implement OpenGL ES 2.0 on top of Direct3D, and the translation does not add significant overhead.

	Desktop GL (fps)	ANGLE (fps)
MapsGL, San Francisco street level http://maps.google.com/mapsgl	32	33
WebGL Field, "lots" setting http://webglsamples.googlecode.com/hg/field/field.html	25–48	25–45
Flight of the Navigator http://videos.mozilla.org/serv/mozhacks/flight-of-the-navigator/	20 minimum	40 minimum
Skin rendering http://alteredqualia.com/three/examples/webgl_materials_skin.html	62	53

Table 39.2. Performance comparison between ANGLE and native OpenGL implementations in sample applications. The results were obtained using Google Chrome 15.0.874.106m on a laptop with a Core i7-620M (2.67 GHz dual core), GeForce 330M, running Windows 7 64-bit. Framerates were determined using FRAPS (http://www.fraps.com/), and vsync was forced off.

39.4 Future Work

ANGLE is continuing to evolve, and there is future work to be done implementing new features, improving performance, and resolving defects. Additional features that could be added include depth textures, wide lines, and multisample EGL configs. Areas for improving performance include target-dependent flipping of rendering, optimizations to texture loading and pixel readback, and bottlenecks as shown by profiling applications. Another possible future direction for ANGLE's development could be to implement a Direct3D 11 back-end. This would allow us to support features not available in Direct3D 9, future versions of the OpenGL ES API, and future operating systems where Direct3D 9 is not ubiquitous.

39.5 Conclusion

This chapter explained the motivation behind ANGLE and described how it is currently used in web browsers both as an OpenGL ES 2.0-based renderer and as a shader validator and translator. We discussed many of the challenging aspects of the implementation of this project, primarily in mapping between OpenGL and Direct3D, and explained how they were mastered. We discussed some of the optimizations we have made in our implementation in order to provide a conformant Open GL ES 2.0 driver that is both competitive in performance and fully featured. The development of ANGLE is ongoing, and we welcome contributions.

39.6 Source Code

The source for the ANGLE Project is available from the Google Code repository.[1] This repository includes the full source for the ANGLE libGLESv2 and libEGL libraries as well as some small sample programs. The project can be built on Windows with Visual C++ 2008 Express Edition or newer.

Acknowledgments We would like to acknowledge the contributions of our TransGaming colleagues Shannon Woods and Andrew Lewycky, who were coimplementers of ANGLE. Thanks also to Gavriel State and others at TransGaming for initiating the project, and to Vangelis Kokkevis and the Chrome team at Google for sponsoring and contributing extensions and optimizations to ANGLE. Finally, we thank the Mozilla Firefox team and other community individuals for their contributions to the project.

[1] http://code.google.com/p/angleproject/

Bibliography

[3Dlabs 05] 3Dlabs. *GLSL Demos and Source Code from the 3Dlabs OpenGL 2 Website.* http://mew.cx/glsl/, 2005 (accessed November 27, 2011).

[ANGLE 11] ANGLE Project. *ANGLE Project Extension Registry*. https://code.google.com/p/angleproject/source/browse/trunk/extensions, 2011 (accessed November 27, 2011).

[Bridge 10] Henry Bridge. *Chromium Blog: Introducing the ANGLE Project.* http://blog.chromium.org/2010/03/introducing-angle-project.html, March 18, 2010 (accessed November 27, 2011).

[Ek 05] Lars Andreas Ek, Øyvind Evensen, Per Kristian Helland, Tor Gunnar Houeland, and Erik Stiklestad. "OpenGL ES Shading Language Compiler Project Report (TDT4290)." *Department of Computer and Information Science at NTNU.* http://www.idi.ntnu.no/emner/tdt4290/Rapporter/2005/oglesslc.pdf, November 2005, (accessed November 27, 2011).

[FSF 11] Free Software Foundation. "Bison—GNU parser generator." *GNU Operating System.* http://www.gnu.org/s/bison/, May 15, 2011 (accessed November 27, 2011).

[MSDN 11a] Microsoft. *Coordinate Systems (Direct3D 9) (Windows).* http://msdn.microsoft.com/en-us/library/bb204853(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11b] Microsoft. *D3DUSAGE (Windows).* http://msdn.microsoft.com/en-us/library/bb172625(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11c] Microsoft. *Direct3D 9 Graphics (Windows).* http://msdn.microsoft.com/en-us/library/bb219837(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11d] Microsoft. *Feature Summary (Direct3D 9 for Windows Vista).* http://msdn.microsoft.com/en-us/library/bb219800(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11e] Microsoft. *Lost Devices (Direct3D 9) (Windows).* http://msdn.microsoft.com/en-us/library/bb174714(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11f] Microsoft. *Point Sprites (Direct3D 9) (Windows).* http://msdn.microsoft.com/en-us/library/bb147281(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11g] Microsoft. *Processing Vertex Data (Direct3D 9) (Windows).* http://msdn.microsoft.com/en-us/library/bb147296(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11h] Microsoft. *ps_3.0 Registers (Windows).* http://msdn.microsoft.com/en-us/library/bb172920(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11i] Microsoft. *Queries (DirectD9) (Windows).* http://msdn.microsoft.com/en-us/library/bb147308(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11j] Microsoft. *Semantics (DirectX HLSL).* http://msdn.microsoft.com/en-us/library/bb509647(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[MSDN 11k] Microsoft. *Viewports and Clipping (Direct3D 9) (Windows).* http://msdn.microsoft.com/en-us/library/bb206341(VS.85).aspx, September 6, 2011 (accessed November 27, 2011).

[Flex 08] The Flex Project. "Flex: The Fast Lexical Analyzer." *Sourceforge.* http://flex.sourceforge.net/, 2008 (accessed November 27, 2011).

[Khronos 11a] The Khronos Group. *Handling Context Lost.* http://www.khronos.org/webgl/wiki/HandlingContextLost, November 17, 2011 (accessed November 27, 2011).

[Khronos 11b] The Khronos Group. "Khronos Native Platform Graphics Interface (EGL Version 1.4)." *Khronos EGL API Registry.* Edited by Jon Leech. http://www.khronos.org/registry/egl/specs/eglspec.1.4.20110406.pdf, April 6, 2011 (accessed November 27, 2011).

[Khronos 11c] The Khronos Group. *OpenGL ES 2.0 Common Profile Specification (Version 2.0.25).* Edited by Aaftab Munshi and Jon Leech. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf, November 2, 2010 (accessed November 27, 2011).

[Khronos 11d] The Khronos Group. *OpenGL Registry.* http://www.opengl.org/registry/, (accessed November 27, 2011).

[Khronos 11e] The Khronos Group. "The OpenGL ES Shading Language (Version 1.0.17)." *Khronos OpenGL ES API Registry.* Edited by Robert J. Simpson and John Kessenich. http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf, May 12, 2009 (accessed November 27, 2011).

[Khronos 11f] The Khronos Group. "WebGL Specification (Version 1.0)." *Khronos WebGL API Registry.* Edited by Chris Marrin. https://www.khronos.org/registry/webgl/specs/1.0/, February 10, 2011 (accessed November 27, 2011).

[TransGaming 11] TransGaming. *GameTree TV: Developers.* http://gametreetv.com/developers, 2011 (accessed November 27, 2011).