



YAPPA Embedded OpenGL Library

2008
YAPPA Corporation



YAPPA Embedded OpenGL Library

1 Library Features

1.1 Implemented standard

The library implements OpenGL/ES 1.1 common profile. In addition there are custom extensions added to this implementation for better control over various optimization features designed into the core of the library.

The following list provides information regarding issues where the library implementation differs from the standard or chooses specific values not specified in the standard:

- Supported rendering resolution is up to 1024x1024 (viewport width and height). Current build is for QVGA resolution only.
- Supported color depth is 16bit 5:6:5 RGB format.
- Maximum number of user defined clipping planes is 1.
- Maximum texture size is 4096x4096.
- Maximum number of mipmap levels is 9.
- Line width is limited to 1.
- Smoothed point size is limited to 1.

1.2 Caching mechanism

The library implements a caching system for application level data (geometry information). This caching system reduces data transfer and format conversion overheads between client application and the library. In addition it enables for unique rendering optimization to be performed during 3D rendering time.

The caching mechanism detects a call to render objects which have been previously stored in the library's cache and uses the cached information. When new data is used to perform rendering it is stored in the libraries cache, possibly replacing older data. The library tracks usage statistics of application data to decide which data should be cached at any given time.

Once objects are cached changing their data at the application level will not affect the draw result. This is because cached object's data which is stored internally within the caching system is used for the actually drawing.

1.2.1 What gets cached

The following data can be cached:

- Vertex coordinates
- Vertex normals
- Vertex colors
- Vertex texture coordinates
- Vertex second texture coordinates

For data to be cached it has to have at least 9 vertices when drawing polygons, 6 vertices when drawing lines, and 3 vertices when drawing points.

This reduces cache lookup and storage overheads for small application objects. Each time a call to *DrawArrays* or *DrawElements* is performed the caching mechanism tries to find the specified data in the cache. If it did not find it might decide to cache the new data (depending on usage popularity).



When *DrawElements* is used the caching system also saves the index information used to access the vertex elements (vertex indices).

1.2.2 Matching cached elements

Cached elements are matched to data passed to the draw function according to the following information:

- Application data type (coordinate, normals, etc.)
- Data pointer
- Data type
- Data element size
- Array stride
- Array element count
- Drawing mode (Triangles, Fan, Lines, etc.)
- First element index (in case of *DrawArrays*)
- Index arrays pointer (in case of *DrawElements*)
- Index arrays data type (in case of *DrawElements*)

A mismatch in one or more of these properties will cause the current information to be potentially cached as a new "object" in the caching system.

1.2.3 Cache system memory usage

The cache systems maximum memory usage definitions are fixed in the library. The library can be built according to specification with different values for these parameters according to the target system's needs.

Parameters:

- Maximum cache system memory size. Current default is 1MB.
- Maximum number of tracked objects (also maximum number of cached objects). Current default is 4096.

1.2.4 Extension API's

Each application data type can be enabled or disabled for caching. This can be controlled during runtime. For example it might be required to disable caching of a specific application data type (for instance mapping coordinates) before a specific call to *DrawArrays* and enable caching for that data type later. The API's to control this are:

glEnableCachingOES(GLbitfield flags)
glDisableCachingOES(GLbitfield flags)

The *flags* parameter is a bit mask specifying which application data types will be enabled or disabled (respectively) for caching.

The available data types are:

GL_CACHE_VERTICES_BIT_OES
GL_CACHE_NORMALS_BIT_OES
GL_CACHE_COLORS_BIT_OES

GL_CACHE_TEXCOORDS_BIT_OES
GL_CACHE_MULTI_TEXCOORDS_BIT_OES

By default all caching flags are enabled.



1.2.5 Buffer Objects

When an OpenGL application uses buffer objects they force enabling of the relevant caching system for each call to a draw function that uses that buffer objects. This ensures caching of the buffer objects.

1.3 Library build options

The library has various build options for removing some of its optimization features according to target system's needs. The following lists the major options for library delivery.

- *DISABLE_CACHING*: Disables the caching mechanism. This reduced memory used for caching and library footprint size.
- *USE_MINIMUM_MEMORY*: Library tries to use less memory by performing minimal preallocation for its internal scratchpad. This might come at the expense performance during application startup.
- *SMALL_CODE_SIZE*: Library is omitted of common usage scenario optimizations and other optimizations which require code duplication to reduce footprint size.
- *HIGH_PRECISION_LOW_RESOLUTION*: Library calculation precision is reduced to support resolutions higher than QVGA. This option should be enabled for target platforms or applications with higher than QVGA rendering targets.

Current library is not built with any special build options. Resulting in a speed optimized version.

1.4 Code generation control

The library implements a function generator feature which creates optimized code for polygon rendering in real time according to the current GL state definition. This enables optimizations unique to specific polygon filling situation, without maintaining thousands of functions to support each possible case. These functions are cached in memory and used each time the GL context definition matches the function's profile.

1.4.1 Extension API's

1.4.1.1 Enable/Disable code generation

Using extension API's it is possible to disable code generation mechanism in runtime. This will cause all polygon drawing to be performed using the generic polygon filler function. This function is much slower as it is the baseline from which all other optimized functions are generated from. To control this use the following API:

```
glEnableCodeGenOES(GLbitfield flags)  
glDisableCodeGenOES(GLbitfield flags)
```

The *flags* parameter is a bitfield mask defining which of the librarie's subsystem supporting code generation we want to enable or disable (respectively). Currently supported subsystems are:

```
GL_CODEGEN_RASTER_BIT_OES
```



1.4.1.2 Manual code generation

It is possible to pre-cache an optimized function for rendering a specific GL context. To do this you must set the GL context using the standard OpenGL/ES API's and then call:

GLint glGenerateCodeOES(GLbitfield flags)

The *flags* parameter is defined in section 1.4.1.1

The return value of the function is an ID identifying the optimized function that was generated for the current context. This ID can be used later on to release the function when we know we will no longer require a function for rendering such contexts in the near future (this can be useful for reducing memory usage). Releasing a generated function is performed by calling:

glReleaseCodeOES(GLbitfield flags, GLint id)

Where the *id* is the ID returned by the *glGenerateCodeOES* function.

Please note that these functions should be used after disabling automatic code generation.

1.4.2 Controlling memory usage

Generated functions are always smaller than the baseline function from which they are all generated because they don't contain the code needed to support the generic case; rather they contain optimized code for a specific case. These functions are cached in memory to be used in the future. Each time an appropriate function is found and called its "call score" is increased so it is less likely to be discarded if memory allocated for generated functions runs low.

Controlling the memory size allocated for this cache may sometimes be helpful, for example reducing generated code cache size can reduce memory requirements of the application while increasing code cache size might enable much higher performance for applications which use more polygon filling functions than the number fitting in the current generated code cache size.

The library can be built according to specification with different values for these parameters.

Parameters:

- Maximum code cache system memory size. Current default is 1MB.
- Maximum number of stored functions. Current default is 256.

1.5 Context and window management

The OpenGL/ES library provides an implementation for rendering images using the OpenGL/ES. What the OpenGL/ES standard does not define is how to create and manage the GL context and how to create and manage output rendering buffers.

These features are normally provided by extensions to the OpenGL/ES library.

The current library implements a minimal such extension than can be used to create a context and output images onto a display device on specific platforms.

Supported platforms are:

- Win32
- ARM versatile development platform

It is important to note that this is a minimal implementation for basic use and evaluation. Further extensions can be easily adopted but require more specific definitions for target usage options.

1.5.1 Extension API's

Context management is not required since the library provides static context per linked process. Any enhancements to the context management will change this in the future.

Following is a list of the extension API's for window management.

- *createEGLWindow(int width, int height, char *name)*
This function should be called once at process startup for creating the output rendering buffer.

width and *height* define windows size, and should coincide to the environments display size definitions.

name is a string name for the defined window.

- *postEGLWindow()*
This function is used to copy the output buffer to the systems display device.
- *void *getWindowBuffer(int what)*
This function is used to retrieve a pointer to the specified output buffer used by the current process. *what* can be any of:
GL_COLOR_BUFFER_BIT – for image output buffer pointer
GL_DEPTH_BUFFER_BIT – for depth buffer pointer
GL_STENCIL_BUFFER_BIT – for stencil buffer pointer
- *closeEGLWindow()*
This function is used for releasing memory allocated by the *createEGLWindow()* function.

2 Performance overview

2.1 Speed

The following table lists speed performance results of some sample application using the GL Library.

The listed frame rates are measured according to the libraries render time. This do not take into account application overhead and frame buffer blitting. This is to reduce biases caused by application overhead and current architecture display hardware access.

Speed results listed here were measured on a 192MHz ARM9 platform.

Sample	Polygons	Fragment types	Lighting	Frame rate (FPS)
San Angeles	30,536	flat shaded	enabled, 3 light sources	8 (avg), varies during playback
Gears	888	Smooth	enabled, 1 light source	50
Duck	617	Smooth	disabled	93
Rose	1516	flat 47%, texture 30%, smooth 23%	disabled	41
Ship	4698	smooth 66%, flat 20%, texture 14%	disabled	18
Chess	12696	98% texture, 2% flat	disabled	16
Magic view	1516	45% flat, 8% texture, 44% smooth	disabled	53



2.2 Memory usage

2.2.1 RAM usage

The libraries RAM usage is built comprised of the following components:

- Frame buffer (including depth and stencil) memory
- State and scratch buffers
- Cache system memory (if caching is enabled)

2.2.1.1 Frame buffer memory

Frame buffer memory requirements are straightforward. A color and depth buffer will always be allocated. Stencil buffer is allocated on first demand.

Color and depth buffer size is 320x240x32bit for a QVGA display (300k).

Stencil buffer size is 320x240x32bit for QVGA display (300k).

2.2.1.2 State and scratch memory

At minimum about 25k is needed to sustain the libraries GL state. Additional memory is allocated for scratch purpose according to the number of vertices passed to the draw commands. The scratch memory grows as needed and extends itself in jumps of 512 elements each time it runs out. This means that if there are no calls to render more than 512 vertices at once it will extend itself only once. Each such extension is about 40k.

For example the Gears sample which renders at most 170 vertices in one draw call will consume a total of 62k for scratch and state management.

The Ship sample with around 5000 vertices requires a total 190k for scratch and state management.

2.2.1.3 Cache system

Cache system memory grows as needed according to the number of detected objects in the application and their sizes. Cache upper memory boundary is fixed and will never be exceeded. Current configuration is 1MB.

For a simple application like the Gears sample with no more than 1000 vertices, the caching system allocates less than 50k.

More complex applications which have more data to be cached will consume more. For example the Ship sample, which includes mapping coordinates, vertex color information, and around 5000 faces, allocated 320k for caching purposes.

2.2.1.4 Code generator memory

Code generation cache memory grows as needed according to the number of generated functions used by the application and their sizes. Upper memory boundary is fixed and will never be exceeded. Current configuration is 1MB.

Typically the memory required for a single generated polygon filler is around 10k (this varies according to the features supported by the generated function).

2.3 Library footprint

Library footprint size can be controlled using different build options. The two most basic configurations are:

- Full library, with all optimization options included is 825k.
- The minimal, without common usage pipeline optimization, code generation and caching mechanism is 284k.