Cryptic Allusion KallistiGL / Programmer's Manual

29th July 2002

Contents

Legal

Chapter 1

Overview

The KallistiGL (KGL) name is not intended to infringe on the OpenGL® trademark owned by Silicon Graphics Inc. It stands for KallistiOS Graphics Library. Please note the following quote (in case you didn't read the legal stuff) from the trademark.pdf published at the SGI website
`http://www.sgi.com/software/opengl/license.html`

> "This product is based on the published OpenGL® API, but is not an implementation which is certified or licensed by Silicon Graphics, Inc. under the OpenGL® API."

We are not making this a library that is going to be 100% OpenGL® complient, nor are we planning get it conformance tested and licensed. It is our goal to make a library that is fast and provides enough features that it will be possible to port some OpenGL® games, and also make it easier for homebrew developers to create new and original works for the Dreamcast. We are striving for a certain level of OpenGL® conformance but do not intend to add features that we believe are either incompatible with an embedded system or don't map easily to the hardware. At the time of this writing KGL is a Dreamcast only API.

Chapter 2

KGL API

This is not an OpenGL® tutorial or complete reference. Those are available online or at a bookstore. This section describes the KGL API currently implemented, the GL_XXX enums supported, and any known non-conformance issues, or DC features for each API function. The API includes non-working stubs for some functions to ease porting. These stubs will be clearly noted. If you think KGL is doing something weird or wrong please refer to this document to determine if the functions in question are supported.

Some of the glKos functions were created to support the Direct Render method of sending polygons to the hardware. Those functions are marker with "(DR)."

2.1 KGL DC-specific APIs

2.1.1 int glKosInit()

The function to intialize KGL. This must be called after KOS and the PVR are intialized. KGL will inherit the active lists specified as parameters to pvr_init. Currently KGL supports the *opaque*, *transparent* and *punchthru* polygon lists. Returns 0 if successful, -1 if no lists are found to be active.

2.1.2 void glKosShutdown()

Call after finishing with KGL. See code example 1, page 7.

2.1.3 void glKosGetScreenSize(GLfloat *x, GLfloat *y)

Get the current video screen size.

2.1.4 void glKosBeginFrame();

Begin frame sequence. See code example 1, page 7.

2.1.5 void glKosFinishFrame()

Finish frame sequence. See code example 1, page 7.

Algorithm 1 DC specific API exmaple

pvr_init_params_t params = {

```
  /* Enable opaque, translucent and punchthru polygons with size 16 */
  { PVR_BINSIZE_16,
    PVR_BINSIZE_0,
    PVR_BINSIZE_16,
    PVR_BINSIZE_0,
    PVR_BINSIZE_16 },
  /* Vertex buffer size 512K */
    512*1024
};
extern uint8 romdisk[];
KOS_INIT_ROMDISK(romdisk);
int main(int argc, char **argv) {
...
    /* Initialize PVR */
    pvr_init(&params);
    /* Call before using KGL */
    glKosInit();
...
    while(foo) {
        ...
        /* Begin frame sequence         */
        glKosBeginFrame();
        draw_op(); /* opaque polys      */
        glKosFinishList();
        draw_tr(); /* transparent polys */
        glKosFinishList();
        draw_pt(); /* punchthru polys   */
        glKosFinishFrame();
        ...
    } /* while */
    ...
    glKosShutdown();
    return 0;
}
```

2.1.6 void glKosFinishList()

Finish with the current list. See code example 1, page 7. XXX - describe lists.

2.1.7 void glKosMatrixIdent() (DR)

Set the DC's matrix regs to identity.

2.1.8 void glKosMatrixDirty() (DR)

Set matrix regs as dirtied. You should do this if you've done any tinkering with the DC's matrix registers outside of using GL calls.

2.1.9 void glKosPolyHdrDirty() (DR)

Set poly header context as dirtied. Similar to the matrix call above, but this one is for the case where you've sent your own primitive headers to the DCPVR2's TA.

2.1.10 void glKosMatrixApply(GLenum mode) (DR)

Apply one of the GL matrices to the DC's matrix regs.

mode
GL_KOS_SCREENVIEW
GL_MODELVIEW
GL_PROJECTION

2.1.11 void glKosSendPolyHdr() (DR)

This function bypasses the normal glBegin( ) / glEnd( ) paradigm and sends the currently *defined polygon header* to the hardware. Where *defined polygon header* is simply the current GL states encapsulated in a *pvr_poly_hdr_t* structure that apply to the object about to be rendered. e.g.: fog, culling, blend functions, texture, current polygon list, etc. Your optimized drawing function is responsible for filling and sending the verticies (pvr_vertex_t) for the object to be rendered by the DCPVR2 hardware.

2.2 Per Fragment Operations

2.2.1 void glBlendFunc(GLenum sfactor, GLenum dfactor)

Only applies when the transparent/punchthru polygon list is active.

Constant	sfactor	dfactor	Computed Blend Factor
GL_ZERO	X	X	(0, 0, 0, 0)
GL_ONE	X	X	(1, 1, 1, 1)
GL_DST_COLOR	X		(R_d, G_d, B_d, A_d)
GL_SRC_COLOR		X	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	X		(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)
GL_ONE_MINUS_SRC_COLOR		X	(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)
GL_SRC_ALPHA	X	X	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	X	X	(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)
GL_DST_ALPHA	X	X	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	X	X	(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)
GL_SRC_ALPHA_SATURATE			not available
GL_CONSTANT_COLOR			not available
GL_ONE_MINUS_CONSTANT_COLOR			not available
GL_CONSTANT_ALPHA			not available
GL_ONE_MINUS_CONSTANT_ALPHA			not available

2.2.2 void glClearDepth(GLclampd depth)

NOP: non-functional. To ease porting.

2.2.3 void glDepthFunc(GLenum func)

The DC supports a number of depth functions.

Supported Depth Functions
GL_NEVER
GL_LESS (default)
GL_EQUAL
GL_LEQUAL
GL_GREATER
GL_NOTEQUAL
GL_GEQUAL
GL_ALWAYS

2.2.4 void glDepthMask(GLboolean flag)

Valid state is GL_TRUE or GL_FALSE. Default state is GL_TRUE. Controls blending of opaque and transparent objects.

2.2.5 glScissor(GLint x, GLint y, GLsizei width, GLsizei height)

The DC uses the PVR USER_CLIP functionality and does not allow fine-grain per-pixel control over the scissor box width, height and placement on the screen. The minimum clip rectangle is a 32x32 area which corresponds with the size of tiles used by the tile accelerator. The PVR swithes off rendering to tiles outside or inside the defined rectangle dependant upon the 'clipmode' bits in the polygon header.

The specified scissor box will always have a size that is some multiple of 32. glScissor(0, 0, 32, 32) allows only the 'tile' in the lower left hand corner of the screen to be modified and glScissor(0, 0, 0, 0) disallows modification to all 'tiles' on the screen. Enabling GL_SCISSOR_TEST or GL_KOS_USERCLIP_OUTSIDE will cause rendered objects to be clipped to the inside or outside of the scissor box respectively.

2.3 Display lists

Not available at this time.

2.4 Drawing functions

Rasterization and per-fragment drawing functions.

2.4.1 void glBegin(GLenum mode)

See the following table for the currently supported primatives. The maximum GL_TRIANGLE_STRIP length is currently limited to 64 vertices.

Mode	Implemented
GL_POINTS	X
GL_LINES	
GL_LINE_LOOP	
GL_LINE_STRIP	
GL_TRIANGLES	X
GL_TRIANGLE_STRIP	X
GL_TRIANGLE_FAN	
GL_QUADS	X
GL_QUAD_STRIP	
GL_POLYGON	

2.4.2 void glColor3f(GLfloat red, GLfloat green, GLfloat blue)

2.4.3 void glColor3fv(GLfloat *v)

2.4.4 void glColor4f(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha)

2.4.5 void glColor4fv(GLfloat *v)

2.4.6 void glColor4ub(GLubyte red, GLubyte green, GLubyte blue, GLubyte alpha)

2.4.7 void glEnd(void)

2.4.8 void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz)

NOP: Stub only.

2.4.9 void glPointSize(GLfloat size)

GL_POINTS are emulated using quads. Points are therefore rendered square. Default size is 1.0 which is 2 pixels square on the DC. Maximum size is 100.0.

2.4.10 void glTexCoord2f(GLfloat s, GLfloat t)

2.4.11 void glTexCoord2fv(GLfloat *v)

2.4.12 void glVertex3f(GLfloat x, GLfloat y, GLfloat z)

2.4.13 void glVertex3fv(GLfloat *v)

2.5 Fog

Uses the PVR2 hardware fog table. Care must be taken when changing fog values because there is only one fog table and any changes made while the PVR is rendering a scene may result in display artifacts. Most noteably if the changes made are large. When vertex fog is implemented this will not be an issue, however vertex fog will take more CPU time.

Parameter name	parameter	default
GL_FOG_MODE	GL_EXP, GL_EXP2, GL_LINEAR	GL_EXP
GL_FOG_DENSITY	density ¿= 0.0	1.0
GL_FOG_COLOR	float [4] color = {R, G, B, A}	{0.5, 0.5, 0.5, 1.0}
GL_FOG_INDEX		Palletted modes not supported
GL_FOG_START	0.0 - 259.0	0.0
GL_FOG_END	0.0 - 259.0	1.0

2.5.1 void glFogi(GLenum pname, GLint param)

```
glFogi(GL_FOG_MODE, GL_EXP2);
```

2.5.2 void glFogiv(GLenum pname, const GLint *params)

2.5.3 void glFogf(GLenum pname, GLfloat param)

```
glFogf(GL_FOG_DENSITY, 0.5f);
```

2.5.4 void glFogfv(GLenum pname, const GLfloat *params)

```
GLfloat fogcolor[4] = {0.5, 0.5, 0.5, 1.0};
glFogfv(GL_FOG_COLOR, fogcolor);
```

2.6 Lighting

This is a decidedly lackluster section. Hopefully lighting will be supported at some point in the future.

2.6.1 void glShadeModel(GLenum mode)

The available modes are GL_FLAT, and GL_SMOOTH. GL_FLAT is default.

2.7 Miscellaneous

2.7.1 void glClear(GLbitfield mask)

NOP: there's nothing to clear on the PVR2. Provided to ease porting.

2.7.2 void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha)

Sets the background color. Alpha parameter has no effect.

2.7.3 void glCullFace(GLenum mode)

Set face culling mode. Valid states are GL_FRONT and GL_BACK. Default state is GL_BACK.

2.7.4 void glDisable(GLenum cap)

Disable GL states. See table in section 2.7.5, on page 12.

2.7.5 void glEnable(GLenum cap)

Enable GL states.

State	Description / Notes	Initial Value
GL_BLEND	**NOP:** will be available once the PVR scene rendering API is completed. For now use glKosFinishList() to switch to the transparent polygon list.	GL_FALSE
GL_CULL_FACE	Polygon culling	GL_FALSE
GL_FOG	PVR table fog	GL_FALSE
GL_KOS_AUTO_UV	Automatically assign texture coordinates to GL_QUAD primatives.	GL_FALSE
GL_KOS_USERCLIP_OUTSIDE	Drawn polys are clipped to the outside of the scissor box.	GL_FALSE
GL_SCISSIOR_TEST	Drawn polys are clipped to the inside of the the scissor box.	GL_FALSE
GL_TEXTURE_2D	2D texturing	GL_FALSE

2.7.6 void glFrontFace(GLenum mode)

Set polygon front face. Valid modes are GL_CW and GL_CCW. Default mode is GL_CCW. *Note: In KGL prior to KOS-1.1.7 the default was GL_CW and could not be changed.*

2.7.7 void glFlush()

NOP: There is no rendering pipeline.

2.7.8 void glHint(GLenum target, GLenum mode)

NOP: It's our way or no way. KGL_DONT_CARE =D Provided to ease porting.

Valid target values	Valid modes
GL_PERSPECTIVE_CORRECTION_HINT	GL_DONT_CARE
GL_POINT_SMOOTH_HINT	GL_FASTEST
GL_LINE_SMOOTH_HINT	GL_NICEST
GL_POLYGON_SMOOTH_HINT	
GL_FOG_HINT	

2.8 Query functions

2.8.1 const GLubyte *glGetString(GLenum name)

Returns a pointer to a descriptive string.

String name
GL_VENDOR
GL_RENDERER
GL_VERSION
GL_EXTENSIONS

2.9 Texture functions

Allowable texture size height and width in pixels are 8, 16, 32, 64, 128, 256, 512, and 1024. You can load a texture using any combination of the previous values. Only 2D textures are supported. Textures are loaded upside down in keeping with the sprit of OpenGL. Be aware of this when mixing KGL and the PVR API. You may have to flip the V texture coordinates (V = 1 - V) depending upon your application.

2.9.1 void glBindTexture(GLenum target, GLuint texture)

The only supported target is GL_TEXTURE_2D.

2.9.2 void glDeleteTextures(GLsizei n, const GLuint *textures)

2.9.3 void glGenTextures(GLsizei n, GLuint *textures)

2.9.4 void glKosTex2D(GLint internal_fmt, GLsizei width, GLsizei height, pvr_ptr_t txr_address,)

If you have already allocated and loaded a texture you can use this shortcut function to make it available to KGL. The *internal_fmt* varies from from the OpenGL spec. The table shows the internal formats supported by the PVR2.

internal_fmt
GL_ARGB1555
GL_RGB565
GL_ARGB4444
GL_YUV422
GL_BUMP
GL_ARGB1555_TWID
GL_RGB565_TWID
GL_ARGB4444_TWID
GL_YUV422_TWID
GL_BUMP_TWID

2.9.5 void glTexEnvi(GLenum target, GLenum pname, GLint param)

Setup the texture environment. See the following table for valid arguments to this function.

target	pname	param
GL_TEXTURE_2D	GL_TEXTURE_ENV_MODE	GL_REPLACE
		GL_MODULATE (default)
		GL_DECAL
		GL_MODULATEALPHA

2.9.6 void glTexImage2D(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels)

Defines a two-dimensional texture. The only supported *target* is GL_TEXTURE_2D. *Border*, *level*, and pixel conversion (*type*) are not supported. The *internalFormat* and *format* arguments must be the same. See the table in section 2.9.4, on page 14.

2.9.7 void glTexParameteri(GLenum target, GLenum pname, GLint param)

Allows specification of additional texture attributes.

target	pname	param
GL_TEXTURE_2D	GL_TEXTURE_FILTER	GL_FILTER_NONE (default) GL_FILTER_BILINEAR
	GL_TEXTURE_WRAP_S	GL_REPEAT (default) GL_CLAMP
	GL_TEXTURE_WRAP_T	GL_REPEAT (default) GL_CLAMP

2.10 Transformation functions

2.10.1 void glDepthRange(GLclampf n, GLclampf f)

We can't access the PVR's depth buffer directly, but the values are used during viewport transformation.

2.10.2 void glFrustum(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat znear, GLfloat zfar)

2.10.3 void glLoadIdentity(void)

2.10.4 void glLoadMatrixf(const GLfloat *m)

Load an arbitrary matrix into the current matrix.

2.10.5 void glLoadTransposeMatrixf(const GLfloat *m)

2.10.6 void glMatrixMode(GLenum mode)

Stack mode	Stack size
GL_MODELVIEW	32
GL_PROJECTION	2
GL_TEXTURE	2

2.10.7 void glMultMatrixf(const GLfloat *m)

Mulitply the current matrix by an arbitrary matrix.

2.10.8 void glMultTransposeMatrixf(const GLfloat *m)

2.10.9 void glOrtho(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat znear, GLfloat zfar)

2.10.10 void glPopMatrix(void)

2.10.11 void glPushMatrix(void)

2.10.12 void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)

As of KOS-1.1.7 the *angle* is in degrees. In prior versions the *angle* was expressed in radians.

2.10.13 void glScalef(GLfloat x, GLfloat y, GLfloat z)

2.10.14 void glTranslatef(GLfloat x, GLfloat y, GLfloat z)

2.10.15 void glViewport(GLint x, GLint y, GLsizei width, GLsizei height)

The default values are automatically set during glKosInit( ) and are determined by the current video mode. You can of course choose your own values and KGL will render to the defined rectangle. The defined area will not be clipped. To clip the defined rectangle you will need to use glScissors( ).

2.10.16 void gluLookAt(GLfloat eyex, GLfloat eyey, GLfloat eyez, GLfloat centerx, GLfloat centery, GLfloat centerz, GLfloat upx, GLfloat upy, GLfloat upz)

2.10.17 void gluPerspective(GLfloat fovy, GLfloat aspect, GLfloat zNear, GLfloat zFar)

Chapter 3

Direct Render

When standard KGL just isn't fast enough for your needs it's time to summon the forces of Direct Render. Direct Render, from this point forward referred to as DR, lets you mix the convenience features of KGL with a number of macros and glKOS functions. When used properly you can expect get 3 to 4 times more polys per frame than KGL alone.

The peformace impact of DR methods are so great that the techniques have been applied to the core KGL drawing functions for a nice performance boost. Some benchmarks taken before and after the conversion are posted below. GL_TRIANGLES were used since they are easy to draw and are the least efficient way draw objects, so any improvements in terms of polygons per second (pps) are pretty impressive. The raw PVR benckmarks are even more impressive.

KGL GL_TRIANGLES (trimark): Before conversion to DR
Beginning new test: 1013 polys per frame (60780 per second at 60fps)
Average Frame Rate: ~59.356900 fps (60128 pps)
Entering PHASE_FINAL
Average Frame Rate: ~60.924400 fps (61716 pps)
Average Frame Rate: ~59.356900 fps (60128 pps)
Average Frame Rate: ~61.187100 fps (61982 pps)
Average Frame Rate: ~59.875900 fps (60654 pps)
Average Frame Rate: ~61.186900 fps (61982 pps)

KGL GL_TRIANGLES (trimark): After conversion to DR
Beginning new test: 3125 polys per frame (187500 per second at 60fps)
Average Frame Rate: ~59.873900 fps (187106 pps)
Entering PHASE_FINAL
Average Frame Rate: ~60.928600 fps (190401 pps)
Average Frame Rate: ~60.924800 fps (190390 pps)
Average Frame Rate: ~59.349700 fps (185468 pps)
Average Frame Rate: ~59.995800 fps (187487 pps)
Average Frame Rate: ~59.875900 fps (187112 pps)

The raw PVR benchmarks for triangles are even more impressive. These benchmarks are from the pvrmark and pvrmark_strip test programs.

PVR pvrmark test:
Beginning new test: 6466 polys per frame (387960 per second at 60fps)

Average Frame Rate: ~59.890300 fps (387251 pps)
Entering PHASE_FINAL
Average Frame Rate: ~61.187100 fps (395635 pps)
Average Frame Rate: ~59.875900 fps (387158 pps)
Average Frame Rate: ~59.873900 fps (387145 pps)
Average Frame Rate: ~60.927500 fps (393957 pps)
Average Frame Rate: ~59.414400 fps (384173 pps)

The DCPVR2 can draw long triangles strips even more efficiently.

PVR pvrmark_strips test:
Beginning new test: 12433 polys per frame (745980 per second at 60fps)
Average Frame Rate: ~57.953400 fps (720535 pps)
Entering PHASE_FINAL
Average Frame Rate: ~55.270300 fps (687176 pps)
Average Frame Rate: ~59.290100 fps (737154 pps)
Average Frame Rate: ~59.004700 fps (733606 pps)
Average Frame Rate: ~60.719300 fps (754923 pps)
Average Frame Rate: ~58.388400 fps (725943 pps)

You're probably sick of benchmarks but I have to show you the results of the pure PVR API in action. This is the output of the pvrmark_strips_direct program:

PVR pvrmark_strips_direct test:
Beginning new test: 40233 polys per frame (2413980 per second at 60fps)
Average Frame Rate: ~59.414300 fps (2390419 pps)
Entering PHASE_FINAL
Average Frame Rate: ~60.924800 fps (2451190 pps
Average Frame Rate: ~59.414300 fps (2390419 pps)
Average Frame Rate: ~59.348800 fps (2387782 pps)
Average Frame Rate: ~59.890300 fps (2409568 pps)
Average Frame Rate: ~59.348800 fps (2387782 pps)

These PVR benchmark test programs were used in the development of the Direct Render methods, macros, and glKos functions described in this chapter. We owe this all the Dan Potters' desire to put 6 transparent spheres on the screen at once. DR can do far more then that. In the words of the wizard himself in response to a question about the GL_TRIANGLE benchmarks:

> 1) Every extra amount of time that you spend transmitting polygons to the TA for rendering is less time you have available for doing things like playing background music, updating AI, etc. So the faster we can make this, the better. That's why I started the whole DR thing anyway – I could have made my graphics stuff fast enough without it, but wouldn't have had any time left over for anything else. Plus if you have extra polys left over then you can go about improving the visual quality of the effects.
>
> 2) It's a lot easier to rack up tons of polygons than you might think. For example, if you have a spaceship with 100 quads in it (not very many, let me tell you! =) and you have 10 of them on screen, that's about a fifth of the available time there if you did nothing else.

3.1 Direct Render sample code

Rather than enumerate all the DR macros and functions right away we'll toss out a couple pieces of sample code so you can see the differences between plain old KGL and KGL using DR. Note these are not complete functional programs, for those I strongly urge you to review the *bubbles demo* under the Dreamcast KGL examples.

XXX KGL sphere quads

XXX KGL/DR sphere DR

3.2 Direct Render API

XXX stuff from matrix.h, pvr.h, gl.h XXX

Chapter 4

About this document

This document was written in a stock LyX 1.1.6fix4 distribution, using no fancy add-ons (so it ought to load ok for you if you want to try it). The PostScript, PDF, HTML, and text versions were generated using the export options in LyX.