Lesson 6 - Introduction to GCM

Summary

This lesson will introduce GCM - The primary graphics library for creating 3D scenes on the Playstation 3. We start simple, by getting GCM to output a coloured triangle. Nvidia's Cg shader language will also be introduced, as it's required for displaying any output in GCM.

New Concepts

GCM, Cg, Accessing the RSX graphics memory, loading shaders, vertices, buffer switching

Tutorial Overview

In this lesson we're going to be performing the graphics equivelent of 'Hello World' - outputting a single coloured triangle. If you've ever used OpenGL, you've probably done this before. As GCM is a much lower-level library than OpenGL, it's a bit trickier to get anything outputting on the screen, even moreso as GCM requires you to use shaders before it'll even display any output! We'll keep things 'simple' this lesson, and not use any of the Playstation's SPUs, so all you'll need is a PPU program, set up in the usual way. We are also going to follow the same class hierarchy model as the Graphics For Games module - with a *GCMRenderer* base class to encapsulate the generic graphics programming functions (equivalent to the *OGLRenderer* class), and derived *Renderer* classes to implement application-specific functionality. For this series of tutorials, you are provided with enough code to get started with - the *GCMRenderer* class, and Playstation specific *Mesh* and *Shader* class, with the tutorial text discussing differences with the OpenGL equivalent classes you are familiar with.

GCM Overview

There are two rendering APIs available for use on the Playstation 3 - an OpenGL ES derived library called psGL, and the Playstation specific GCM library. Sony recommend that you use GCM, which is a low level API specifically designed around the hardware capabilities of the Playstation's graphics hardware, and so can more efficiently control the graphics output - psGL is itself built 'on top' of GCM anyway, so we may as well target the custom API. So, how does GCM compare with OpenGL? Well, like OpenGL it uses matrices to perform vertex transformation, and uses vertices, textures, and shaders to render objects to screen. *Unlike* OpenGL 1 and 2, it has no 'immediate mode' - you can't just send vertices using a **glVertex3f**-like command, you must use vertex buffers for all of your vertex rendering. There's also no concept of a matrix stack in GCM, so there's no pushing and popping matrices. These two differences may not affect you too much if you have programmed in OpenGL 3+ core profile, which removes the matrix statck and immediate mode rendering. The biggest difference between GCM and OpenGL is that GCM expects you to do much more memory management yourself. All it gives you is a pointer to the start of useable graphics memory - there's no 'behind the scenes' management of texture memory and vertex buffers. Also unlike OpenGL with its choice of shader languages, in GCM you have only one - **Cg**. This is Nvidia's own shader language, and is syntactically fairly similar to GLSL - if you can use GLSL you should have no trouble with Cg. Explaining all of the Cg concepts is beyond the scope of these tutorials, so if you want to learn all of the ins and outs of Cg, you might like to take a look at the following website:

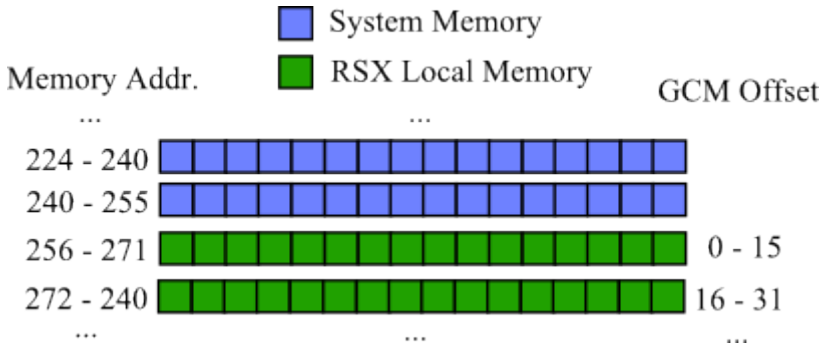
`http://developer.nvidia.com/object/cg_tutorial_home.html`

The website has a complete e-book introducing Cg, and contains plenty of example shaders to produce some interesting graphical effects.

As with OpenGL, the commands you send to GCM via it's API aren't executed immediately, but placed in a *command buffer* to be executed in sequence. As with most other things in GCM, we'll have to allocate the memory for this command buffer ourselves, on system memory rather than the graphics memory.

GCM Memory

Earlier, it was mentioned that GCM does no memory management for you - all you get is a pointer to the start of usable RSX memory. That means we're going to have to create some functions to allocate graphics RAM for things like textures and shaders. GCM calls graphics RAM 'local memory' - as it's local to the RSX processor of the Playstation 3. Like a lot of other parts of the PS3, certain GCM functions require structures to begin at a certain byte alignment, so we must provide functions to allocate aligned memory, too. In addition, rather than memory pointers, some GCM functions use a memory *offset* - how far from the start of graphics memory an address is. Luckily, GCM includes a function to work out an RSX offset address for a pointer.



An example of a GCM offset (NB: Addresses are abstract examples)

GCM Surfaces

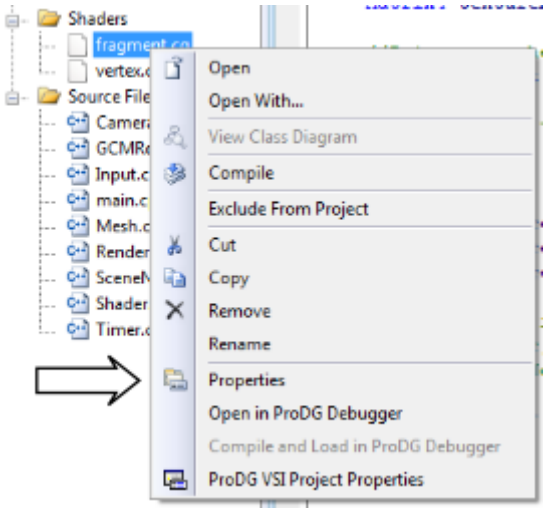
GCM has no default 'built-in' automatic front and back buffer, so you must manually assign some local memory to be the front, back, and depth buffers. In order to write graphical data to this memory, they must be assigned to GCM *surfaces*. These surfaces are conceptually similar to *Frame Buffer Objects* in OpenGL - each surface can have multiple colour buffers and a depth buffer attached to it, just like an FBO. So, in the base *GCMRenderer* class, there are two GCM surfaces - one each for the front and back buffers. Each frame the renderer will swap which of our two surfaces it is rendering to, and display the other.

Compiling Cg Shaders

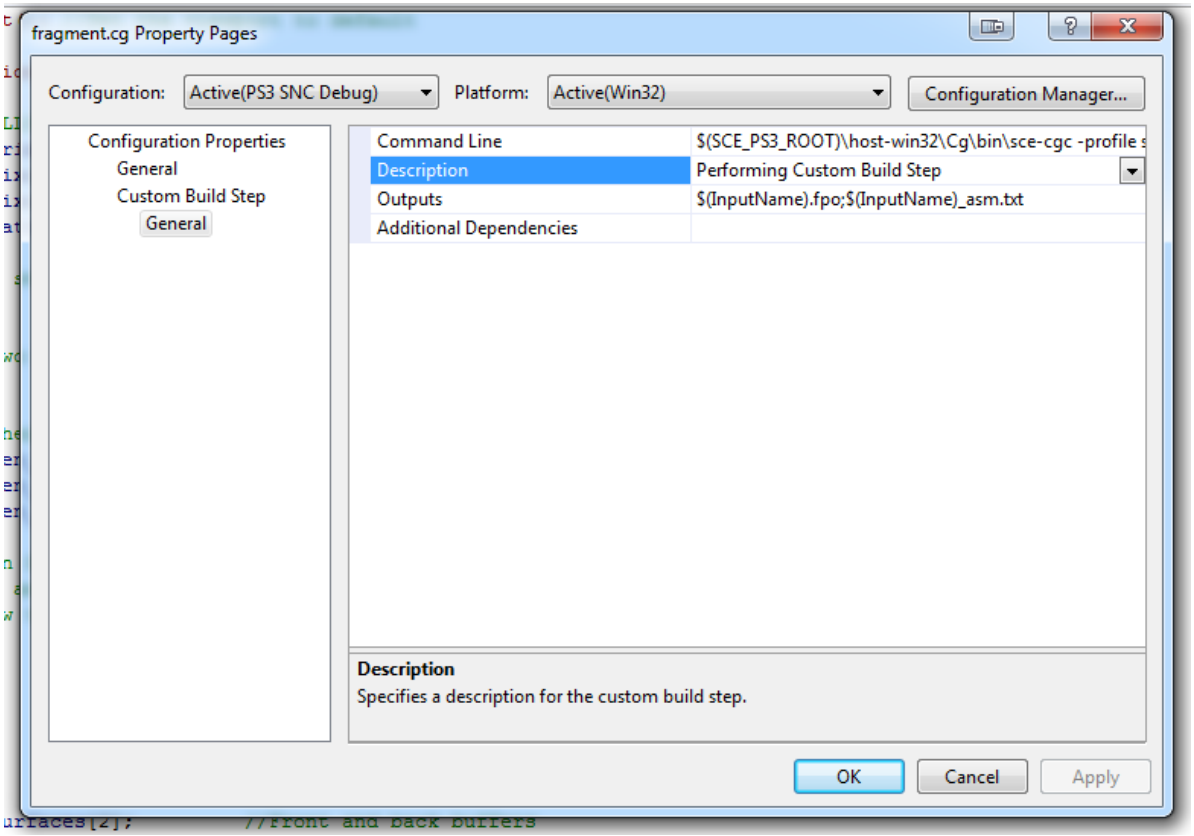
When we were making OpenGL shaders, we could simply provide text files containing out shaders to the OpenGL API, which would compile them into code suitable for running on whatever our graphics hardware is. This is a good idea when we don't know what the underlying hardware of the device we run our OpenGL applications on is, but in the case of the Playstation, we know it will always be the RSX graphics processor. This means the time spent compiling the shaders at runtime is an unnecessary waste, as is even having the code to compile the shaders in memory! Instead, it is common to compile the shaders into a binary at the same time the main program code is compiled, and instead loading the raw binary data of the compiled shader program into memory at runtime.

We can make the compilation of the shaders we write automatic, using a Visual Studio feature called the Custom Build Step. This allows us to define what happens to individual files when we compile a project. To do this, we should add a shader's source files to the Visual Studio solution - you might want to make a VS filter to separate the shaders from your game code.

Once this is done, as well as allowing us to edit the shaders from within Visual Studio, we can also then *right click* on a shader file, and select the *properties* option.



This brings up the properties pane for the file. By selecting *Custom Build Step* from the *Configuration Properties* tab, we can define a command line program call, a text description that will be outputted to the Visual Studio output window during compilation, and an Output file to let Visual Studio know what files to expect out of the build process.



So what program should we call from the command line to compile our shaders? In the SDK installation, there's a Cg compiler written specifically for the Playstation 3's graphics hardware. We can call this program with a couple of command line options to tell the compiler the shader profile to use (either vertex or fragment), where to find the shader, and the name of the file to output.

For vertex shaders we should use the follow command line:

$(SCE_PS3_ROOT)/host-win32/Cg/bin/sce-cgc -profile sce_vp_rsx -o "$(InputName).vpo" "$(InputPath)"

Where *sce-cgc -profile sce_vp_rsx* defines that the shader we wish to compile is a vertex shader, "$(InputName).vpo" (where vpo is short for *vertex program object*) defines the output, and "$(InputPath)" is where the file can be found. The macros $(InputName) and $(InputPath) are made by Visual Studio, and will match up to the filename, and file location, of the file we are editing the properties for.

Similarly, to define the command line for a fragment shader, we should use:

$(SCE_PS3_ROOT)/host-win32/Cg/bin/sce-cgc -profile sce_fp_rsx -o "$(InputName).fpo" "$(InputPath)"

Note the different profile name, and the slightly different filename.

The Cgc compiler outputs a raw binary, containing the program of our shader in a format that can be loaded and ran directly on the RSX processor. This makes the *Shader* class for the Playstation a bit different to the one introduced in the Graphics For Games module, as we will see shortly.

Simple GCM Rendering

To get you started with rendering using GCM you are provided with some basic classes, much as you were during the graphics module. So, as well as the joypad input class you looked at in a previous tutorial, we also have a *Mesh* class, *Shader* class, and *GCMRenderer* class, all tied together into a simple example program that renders a single textured quad (we'll leave discussing the specifics of texturing until the next tutorial, though!). The Playstation SDK comes with the vector, quaternion, and matrix classes you'll need to do perform graphical rendering, so the framework for this tutorial series doesn't include any. The classes are very similar to what you are used to, and even include functions for creating perspective and orthographic matrices, as well as inversion and transposition.

Program outline

Our simple GCM program is going to perform the following steps:

Step 1) Initialise GCM, and our display
Step 2) Load in our precompiled shaders
Step 3) Initialise our textured quad as a vertex buffer
Step 4) Render the scene until we press Start on the joypad
Step 5) Clean up and exit

As usual, there's a bit more to it than first appears, but everything we need to do in each stage will be explained in detail.

Linker Input Dependencies

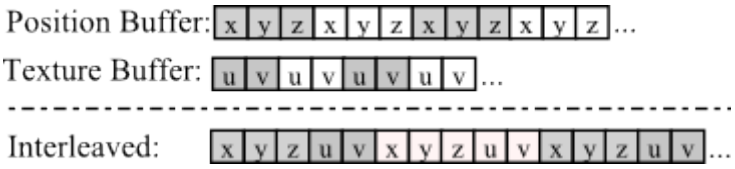
As we'll be using some external libraries in this lesson, in the form of GCM and Cg, we need to let the compiler know where they are. Make sure you have all of the following libraries added in the *additional dependencies* tab of the Visual Studio Configuration properties.

libcgc.a -lgcm_cmd -lgcm_sys_stub -lsysmodule_stub -lm -lsysutil_stub -lio_stub

Mesh class

As part of the small code framework for this tutorial series, we have another version of the *Mesh* class, which as you would expect, encapsulate the functionality required to render geometry using GCM. It is very similar to the class you wrote in the Graphics For Games module, but there are some key differences that should be examined in detail.

In OpenGL, we used a separate vertex buffer for each component in our vertices - one for positions, one for texture coordinates, one for colours, and so on for each vertex *attribute*. We can do so in GCM too if we like, but we can gain a slight performance increase by instead using an *interleaved* vertex buffer; that is, one that has all of the different vertex components together, in a single long vertex buffer.



Comparison between separate VBOs (top) and interleaved VBOs (bottom)

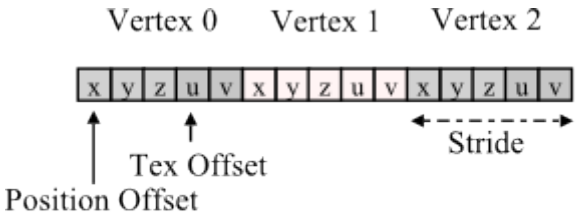
In order to deal with this interleaved data efficiently, rather than have separate arrays of vertex data as member variables of the *Mesh* class, there is instead a single array of a *Vertex* data structure defined in the *Mesh* class header, which looks like this:

```
1 struct    Vertex    {
2    float x, y, z;
3    float u,v;      //Tex Coords
4    //float nx, ny, nz;  //Normals
5    uint32_t rgba;
6 };
```

Renderer.h

By initialising arrays of this **struct**, we get contiguous sections of memory, with the vertex components sequentially interleaved - the position, colour and texture coordinate data of vertex *n* is packed together before the data of vertex *n+1*.

In order to enable GCM to access this interleaved vertex buffer, we need to know the offset into graphics memory to the start of each attribute of the interleaved array, and the buffer stride (how much a pointer advance by to get to the same component of the next vertex - essentially the size of the *Vertex*):



Vertex *n* Tex Pointer = Tex Offset + (Stride * *n*)

Comparison between separate VBOs (top) and interleaved VBOs (bottom)

By setting the size of a vertex, GCM can jump to the start of any vertices data during rendering, and knowing the offset into that vertex of each component will allow it to send the correct attribute data to the shader. We will come to setting the size of a vertex in the next function description, where we actually draw some geometry, but to define the offsets, we can use the API function call *cellGcmAddressToOffset*, which calculates the offset into graphics memory of a pointer. So to calculate the offset for the first position, and first texture coordinate of a **Vertex**, we can do this:

```
1    cellGcmAddressToOffset(&m->vertexData->x,
2                &m->vertexOffsets[VERTEX_POSITION]);
3    cellGcmAddressToOffset(&m->vertexData->u,
4                &m->vertexOffsets[VERTEX_TEXCOORD]);
```

Vertex Data Offsets

The first parameter to *cellGcmAddressToOffset* is the pointer to some data to calculate the memory offset of, and the second parameter is where to put the result - In OpenGL we stored the name of each vertex buffer stream in an array, but in GCM we instead store offsets to each attribute in an array (*vertexOffsets*), so that we only have to calculate them once. These offsets are best defined when the mesh geometry is created, so there are examples of this offset creation in the *GenerateTriangle* and *GenerateQuad* functions in the provided *Mesh* class.

When it comes to actually drawing some geometry, we must use another GCM function, *cellGcmSetVertexDataArray*. This tells GCM about the structure of the *Vertex* data being used to render geometry, and the size of each component in the vertex. For example, to set how the shader accesses vertex data, we might do something like this once we have created some vertex data:

```
1    cellGcmSetVertexDataArray(
2       vertexShader.GetAttributeIndex(VERTEX_POSITION),
3       0,
4       sizeof(Vertex),
5       3,
6       CELL_GCM_VERTEX_F,
7       CELL_GCM_LOCATION_LOCAL,
8       (uint32_t)vertexOffsets[VERTEX_POSITION]
9    );
```

Mesh Class Code Snippet

The first parameter gets which vertex input location the data is pertaining to - in this case the vertices position. Much like we use the *MeshBuffer* **enum** to tie shader vertex attributes and vertex buffer objects together in the OpenGL *Mesh* class we wrote, In the GCM *Mesh* class we use the *VertexAttributes* **enum** to bind vertex shader attributes and vertex offsets together. The third parameter sets the stride of the data - how much to advance by each Vertex, which in this case is the result of the sizeof operator on our Vertex struct. Our positions have 3 floats (lines 5 and 6) and in our simple framework we'll always be storing them in the playstation 3's graphics memory for performance (line 6 - note how it is referred to as *local* memory).

GCM supports rendering via indices, and just like in OpenGL this is done using a slightly different function call than non-indexed rendering. For example, here is the code in the *Mesh* class *Draw* function to render the actual geometry:

```
1    if(vertexOffsets[VERTEX_INDEX]) {   //If we have indices
2       cellGcmSetDrawIndexArray(type, numIndices,
3          CELL_GCM_DRAW_INDEX_ARRAY_TYPE_16,
4          CELL_GCM_LOCATION_LOCAL, vertexOffsets[VERTEX_INDEX]);
5    }s
6    else{
7       cellGcmSetDrawArrays(type, 0, numVertices);
8    }
```

Mesh Class Draw Function Snippet

This should probably feel pretty familiar to you, as it is very similar to how we used OpenGL's functions. The non-indexed draw call, *cellGcmSetDrawArrays*, takes identical parameters to its OpenGL equivalent *glDrawArrays* - *type* defines the primitive type, and also takes in the number of actual vertices to draw. There's a few more parameters to the indexed drawing function compared to

the OpenGL equivelent; instead of a **sizeof** to determine how large each index is, there are specific symbolic constants - CELL_GCM_DRAW_INDEX_ARRAY_TYPE_16 for **unsigned shorts**, and CELL_GCM_DRAW_INDEX_ARRAY_TYPE_32 for **unsigned ints**. As we have direct memory access in GCM, we must also tell the function what type of memory it is in (system or local), and give it either a **pointer**, or if it is in graphics memory (which it should be for performance reasons), an offset.

Shader class

The *Shader* class for this tutorial series is a bit different to that introduced during Graphics For Games. For one, this time a *Shader* represents a single *Shader object* - this means that we have a separate *Shader* for each of the vertex and fragment shaders we use in our program, and so our *GCMRenderer* class *SetCurrentShader* function takes *two* parameters. What's more, we need a few vertex and fragment shader specific functions, so as well as a *Shader* class, we have derived *VertexShader* and *FragmentShader* classes, too.

LoadBinaryShader Function

As we are dealing with precompiled binary shaders, rather than ones loaded in from a text file, it is worthwhile going over how this is achieved by the new *Shader* class. The shader binary is loaded in via the *LoadBinaryShader* function, which takes in a simple **string** for a filename, and opens an **ifstream** to read it - note that we must use the **SYS_APP_HOME** macro to ensure we load the file from the correct place for the current program.

```
1  void  Shader::LoadBinaryShader(std::string name) {
2     name = SYS_APP_HOME + name;
3
4     std::ifstream  file(name.c_str(),std::ios::in|std::ios::binary);
5
6     if(!file) {//Oh no!
7        std::cout << "Can't find file: " << name << std::endl;
8        return;
9     }
```

Shader.cpp

Between lines 10 and 12, we work out how large the file we are loading is going to be, by seeking to the end of the file (line 10), getting the current position of the read pointer (line 11), and then seeking back to the beginning of the file (line 12). We can then use this data size to allocate some memory to store the contents of the shader file (line 14), and then read it in (line 16), and close the file (line 17). On line 19, we cast the resulting data to the **CGprogram** type - this is a Cg specific pointer to represent a shader program, and is pretty much analagous to the program name used in OpenGL. We can then initialise the program ready to use by calling the GCM API function *cellGcmCgInitProgram*.

```
10    file.seekg(0, std::ios::end);
11    unsigned int dataSize = file.tellg();
12    file.seekg(0, std::ios::beg);
13
14    char* data = (char*)malloc(dataSize);
15
16    file.read(data,dataSize);
17    file.close();  //Done with the data, close the file.
18
19    program     = (CGprogram)(void*)data;
20
21    cellGcmCgInitProgram(program);
```

Shader.cpp

To finish off, we need to get another pointer, this time into a specific location inside the shader binary we have just loaded - the *microcode* starting point. Think of this as being the pointer to the first line of the main function inside the shader. GCM has a function to calculate this pointer position, and determine how large the code is, *cellGcmCgGetUCode*, which we call on line 25. Using this information, we can initialise some memory on the GPU-side of the Playstation's memory. Vertex shaders don't need to be in GPU memory to work, but fragment shaders do, so we'll keep things easy and just always keep the shader microcode (the *u* is shorthand for the greek symbol *mu*, used to denote *micro*) in GPU memory. Fragment shaders must also be 64 byte aligned in GPU memory, so again we just always use this to be on the safe side.

```
22    unsigned int ucodeSize;
23    void *ucodePtr;
24
25    cellGcmCgGetUCode(program, &ucodePtr, &ucodeSize);
26    ucode = GCMRenderer::localMemoryAlign(64,ucodeSize);
27    memcpy(ucode, ucodePtr, ucodeSize);
28    cellGcmAddressToOffset(ucode, &offset);
29 }
```

Shader.cpp

Earlier it was mentioned that we must sometimes use *offsets* rather than pointers. It is the case here, with the pointer to the microcode - we actually need the offset to the microcode for the fragment shader, so we call the provided GCM function *cellGcmAddressToOffset* to determine this value. It takes in a **pointer** (the **void*** *ucode* in this case), and places the answer inside the second parameter, a **reference** to an **unsigned int**. Once this is done, we have everything we need for GCM to use a shader - a *CGProgram* parameter, a pointer to the microcode for vertex shaders, and an offset to the microcode for fragment shaders.

As well as binding shaders, we must also occasionally set uniform data on them, to control what their output will be. If you remember, the method for doing this in OpenGL was somewhat long winded, requiring a call to *glGetUniformLocation* to find the location of a given named uniform, and then one of a number of calls to **uniform** setting functions, such as *glUniformMatrix4fv* and *glUniform1i*. GCM does away with most of that, and along with a single uniform 'get' function, has only three uniform parameter setting functions - one for setting texture samplers (which we will look at in the next lesson), and one each for vertex and fragment shaders. In GCM, sending incorrect data with these functions is very dangerous, and will quite likely lead to the program crashing, so we wrap these functions inside our own *Shader* class functions to try and prevent this from happening.

First off, lets take a look at the *Shader* class *GetParameter* function, analogous to the *glGetUniformLocation* function in OpenGL. In our *Shader* class, we actually keep a map, which maps strings to a special **CGparameter** data structure, which is the identifer for a uniform datatype in a shader. By storing the results of our *GetParameter* requests in a map, we can keep track of our parameters in such a way that we don't have to keep asking GCM where our uniforms are, which might be a performance penalty. So, on line 31, we try to find a value in our map that equates to the incoming parameter name (line 31), and if we find it, just return it (line 32-34).

```
30 CGparameter Shader::GetParameter(std::string name) {
31    std::map<std::string,CGparameter>::iterator i = uniforms.find(name);
32    if(i != uniforms.end()) {  //if its in the map, return it!
33       return i->second;
34    }
```

Shader.cpp

If we *don't* find it, we instead use the GCM function *cellGcmCgGetNamedParameter*, that looks for a given parameter name in a given program, and returns a *CGparameter* data structure. If this value is 0, then the parameter is not in the shader, and outputs a warning message. Either way, both the parameter and its name are inserted into the map, so we can find it quicker later on.

```
35    CGparameter p = cellGcmCgGetNamedParameter(program, name.c_str());
36    if(!p) {
37        std::cout << "Can't find named parameter:" << name << std::endl;
38    }
39
40    uniforms.insert(std::make_pair(name,p));
41    return p;
42 }
```

Shader.cpp

Using this function, we can now *get* the uniforms of our shaders, but we still need to be able to *set* them! This is simply handled by a *SetParameter* function for the *VertexShader* and *FragmentShader*, which looks like this for vertex shaders:

```
43 void  Shader::SetParameter(std::string name, float*data) {
44    CGparameter p = GetParameter(name);
45    if(p) {
46        cellGcmSetVertexProgramParameter(p, data);
47    }
48 }
```

Shader.cpp

The function for Fragment shaders is slightly more involved, requiring the *CGprogram* and the offset of the fragment shader:

```
49 void  Shader::SetParameter(std::string name, float*data) {
50    CGparameter p = GetParameter(name);
51    if(p) {
52        cellGcmSetFragmentProgramParameter(program, p, data, offset);
53    }
54 }
```

Shader.cpp

We simply use our *GetParameter* function to get the required *CGparameter*, and then use the *cellGcmSetVertexProgramParameter* (or *cellGcmSetFragmentProgramParameter*) function to set the uniform data to the data pointed to by the second input parameter. Note how we check that the CGparameter actually exists before trying to set data to it - although attempting to set non-existing uniforms is generally well handled in OpenGL, it will result in **bad** things happening inside GCM! The internal workings of GCM will work out how large any given uniform datatype is, and only send that many floats worth of data to the shader, and so only requires a single function, no matter what the datatype.

There is unfortunately a special case when setting uniforms - *matrices*. While the inbuilt matrix class of GCM is like the *Matrix4* class we are used to using with OpenGL, in that it is column major (the first 4 data elements define the first column of the matrix), Cg, the shading language we are using, expects its shaders to be in row major order (the first 4 data elements define the first row of the matrix). This is quite annoying, and we'd have to do lots of flipping around of matrices by transposing them (which flips a matrix along its diagonal), which effectively turns rows into columns and vice versa.

So instead, we overload the *SetParameter* function, with an additional function that takes a **Matrix4** instead of a **float pointer**. It looks like this:

```
55 void  Shader::SetParameter(std::string name, Matrix4 &totranpose) {
56    Matrix4 tempMatrix = transpose(totranpose);
57    SetParameter(name, (float*)&tempMatrix);
58 }
```

Shader.cpp

Not a lot to it! All it does is create a temporary matrix that equates to the transpose of the incoming *Matrix4*, and then uses our other *SetParameter* function to set the actual data, by casting the temporary matrix to a pointer to some floating point data - if you cast your mind back to the Graphics For Games module, you will recall that you did the same case in OpenGL when sending matrices to a shader, too.

The last function we have for setting uniform parameters is one specific to fragment shaders. If a uniform datatype is changed on a fragment shader, then we must manually clear the instruction cache of the fragment shader - or we run the risk of the shader continuing to use old data in its calculations. While we *could* just clear the cache inside the **SetParameter** function, that might lead to a performance hit - setting 10 uniforms will clear the cache 10 times, whereas if we keep it a separate function, we can call it after we are done changing uniforms, and only have to clear the cache once.

```
59 void  FragmentShader::UpdateShaderVariables() {
60    cellGcmSetUpdateFragmentProgramParameter(offset);
61 }
```

Shader.cpp

The last shader specific function to look at is the *SetDefaultAttributes* function of the vertex shader. Just as with OpenGL, we need a way of tying our vertex buffer data to the input parameters of our vertex shaders. The process is a bit more involved than with GLSL shaders in OpenGL, but not by much. We get the parameters to our vertex input data in the same way as we do uniform data, using *cellGcmCgGetNamedParameter*. We could use the *GetParameter* function described earlier to do this, but doing so would put the parameter result in the map, which should be unnecessary, as the only time we will ever look for these parameters is in this function. So instead, on lines 28, 60 and 62, we call the GCM function with the appropriate vertex input data name. Then, we must turn these parameters into resources - essentially these represent where in the actual rendering hardware will a particular parameter be bound to. We store the results of this in an array of such resources, just as we do with the *bufferObject* array in the OpenGL *Mesh* class.

```
62 void VertexShader::SetDefaultAttributes() {
63    CGparameter position_param =
64                cellGcmCgGetNamedParameter(program, "position");
65
66    CGparameter colour_param   =
67                cellGcmCgGetNamedParameter(program, "color");
68
69    attributes[VERTEX_POSITION]   =
70       cellGcmCgGetParameterResource(program,position_param)- CG_ATTR0;
71
72    if(colour_param)  {
73       attributes[VERTEX_COLOUR]  =
74       cellGcmCgGetParameterResource(program,colour_param)  - CG_ATTR0;
75    }
76 }
```

Shader.cpp

GCMRenderer Class

We now know how to manipulate vertex data, and how to load in and set uniforms on shaders - but we still need a way of encapsulating GCM as a whole. That's where the *GCMRenderer* class comes in. Much like the *OGLRenderer* class you are used to, the *GCMRenderer* encapsulates the functionality required to initiate graphics rendering on the Playstation, as well as how to handle graphics memory, and keep track of the model, view and projection matrices.

Let's take a look at the **constructor** for this class:

```
#define COMMAND_SIZE    (65536)         //64KB
#define BUFFER_SIZE     (1024*1024)     //1MB

GCMRenderer::GCMRenderer(void)   {
   if(cellGcmInit(COMMAND_SIZE, BUFFER_SIZE,
                  memalign(1024*1024, BUFFER_SIZE))!= CELL_OK) {
      std::cout << "cellGcmInit failed!" << std::endl;
   }

   camera   = NULL;
   root     = NULL;

   InitDisplay();
   InitSurfaces();
}
```

GCMRenderer.cpp

The GCM library itself is initialised with a single function call - *cellGcmInit*. It takes in 3 parameters - a default command buffer size, a total buffer size, and a pointer to the start of the total buffer. The default command buffer and the total buffer must be at least 64KB and 1MB, respectively - the values we assigned to the COMMAND_SIZE and BUFFER_SIZE macros earlier. To allocate the memory for our command buffer, we use the **memalign** function - as well as being at least (and in fact must be a multiple of) 1MB, the buffer must be 1-megabyte aligned, so we can't just use malloc. These buffers are used to store the list of commands to be sent to the RSX for processing - the GCM functions called throughout the renderer will result in commands being pushed onto this command list, which the RSX will pop off as it consumes and completes instructions.

The GCMRenderer is based on familiar design pattern - we have a *Camera* class to control the view matrix, and a *SceneNode* hierarchy root to control what is being rendered. Although we have now initialised GCM, we have not yet enabled any display output, or created any of the GCM surfaces we need to render into - these are both created in the next two function calls.

Next up, we have *InitDisplay* - this will turn on the display output, and set the screen resolution, too. The first thing this function does is attempt to set the output resolution, using the *SetResolution* function. The resolution chosen will depend on the type of monitor or television the Playstation is connected to, and the resolution set in the Playstation 3's XMB user interface. It may even be the case that a lower resolution than the 'native' resolution is desirable, in cases where the fragment shader is so costly to run it impacts the framerate - a lower resolution means less fragments are generated! For this simple framework though, we'll just try and enable the highest valid resolution we can - the *SetResolution* function returns **true** if a given resolution selection has been enabled, or **false** otherwise.

```
void GCMRenderer::InitDisplay() {
   std::cout << "Initialising display!" << std::endl;

   if(SetResolution(GCM_RESOLUTION_1080)) {
      std::cout << "Using 1080p" << std::endl;
   }
```

```
7     else if(SetResolution(GCM_RESOLUTION_720)) {
8        std::cout << "Using 720p" << std::endl;
9     }
10    else if(SetResolution(GCM_RESOLUTION_576)) {
11       std::cout << "Using NTSC" << std::endl;
12    }
13    else if(SetResolution(GCM_RESOLUTION_480)) {
14       std::cout << "Using PAL" << std::endl;
15    }
16    else {
17       std::cout << "No valid resolution available!" << std::endl;
18       return;
19    }
```

GCMRenderer InitDisplay Function

Once the resolution has been set, we can use a couple of Playstation SDK functions to get the current video output state (line 23, into the struct on line 20), and the current output resolution (line 24, using the previously defined struct to fill in a new resolution struct on line 21). From the result of this, we can then get the correct screen width, height, and ratio (lines 27-29) to use for generating our screen sized buffers.

```
20    CellVideoOutState          vid_state;
21    CellVideoOutResolution     resolution;
22
23    cellVideoOutGetState(CELL_VIDEO_OUT_PRIMARY, 0, &vid_state);
24    cellVideoOutGetResolution(vid_state.displayMode.resolutionId,
25       &resolution);  //Query the current state of the video resolution
26
27    screenWidth       = resolution.width;
28    screenHeight      = resolution.height;
29    screenRatio       = (float)screenWidth / (float)screenHeight;
```

GCMRenderer InitDisplay Function

Then, we complete setting up the Playstation's video settings (lines 31 - 37), and enable v-sync (line 38). V-sync links the maximum framerate to the refresh rate of the screen it is attached to - generally this means we won't get more than 60 frames per second. Lastly, we use one final GCM inbuilt **struct** and function, in order to determine where in memory the start of usable graphics memory is - we then store the pointer inside a member variable of the *GCMRenderer* class, using the *setLocalMem* function.

```
30    //Configure the video output...
31    CellVideoOutConfiguration  video_cfg;
32    memset(&video_cfg, 0, sizeof(CellVideoOutConfiguration));
33    video_cfg.resolutionId  = vid_state.displayMode.resolutionId;
34    video_cfg.format     = CELL_VIDEO_OUT_BUFFER_COLOR_FORMAT_X8R8G8B8;
35    video_cfg.pitch         = screenWidth*4;
36
37    cellVideoOutConfigure(CELL_VIDEO_OUT_PRIMARY,&video_cfg, NULL, 0);
38    cellGcmSetFlipMode(CELL_GCM_DISPLAY_VSYNC);
39
40    CellGcmConfig config;
41    cellGcmGetConfiguration(&config);
42    setLocalMem((uint32_t)config.localAddress);
43 }
```

GCMRenderer InitDisplay Function

The *SetResolution* function called in *InitDisplay* is pretty simple. It takes in an enumerated value that equates to a list of supported resolutions, and then queries the hardware as to the suitablility of this resolution for the current connected display device (line 48), and if so, gets the resolution settings (such as width etc) and stores them in a struct (on line 53). Using the resolution and its information, a *CellVideoOutConfiguration* **struct** is filled - this is used to then configure the actual video hardware to the requested resolution (line 65). Hopefully, we'll then be able to return **true**, otherwise **false** will be returned - that's why we have the **else-if** block in the previous function.

```
44 bool GCMRenderer::SetResolution(GCMResolution resolution) {
45    //First, query if resolution is viable
46    CellVideoOutResolution resInfo;
47
48    if(!cellVideoOutGetResolutionAvailability(CELL_VIDEO_OUT_PRIMARY,
49                        resolution, CELL_VIDEO_OUT_ASPECT_AUTO,0)) {
50       return false;
51    }
52    if(cellVideoOutGetResolution( resolution, &resInfo ) != 0) {
53       return false;
54    }
55
56    CellVideoOutConfiguration config = {   resolution,
57       CELL_VIDEO_OUT_BUFFER_COLOR_FORMAT_X8R8G8B8,
58       CELL_VIDEO_OUT_ASPECT_AUTO,
59       {0,0,0,0,0,0,0,0,0},
60       cellGcmGetTiledPitchSize( resInfo.width * 4 )
61    };
62
63    if(cellVideoOutConfigure( CELL_VIDEO_OUT_PRIMARY, &config,
64                              NULL, 0 ) != 0) {
65       return false;
66    }
67    return true;
68 }
```

GCMRenderer InitDisplay Function

In the *InitDisplay* function, we also saw the first mention of local memory - memory directly attached to the RSX processor. As mentioned earlier, when we initialise GCM, we get a pointer to the start of available graphics memory, and from then on must handle memory allocation ourselves. There are three functions in GCMRenderer that deal with this memory allocation - one to set the start of graphics memory, one to allocate a block of memory, and one to allocate an *aligned* block of memory. Lets take a look at each of these functions in turn, starting with the simplest, *setLocalMem*.

```
1 void GCMRenderer::setLocalMem(uint32_t to) {
2    localHeapStart = to;
3 }
```

GCMRenderer setLocalMem Function

Not much to it - all it does is set a member variable to the input parameter. Next up is localMemoryAlloc, which performs a similar function as **malloc** or **new**, for RSX memory:

```
1 void* GCMRenderer::localMemoryAlloc(const uint32_t size) {
2    uint32_t currentHeap = localHeapStart;
3    localHeapStart      += (size + 1023) & (~1023);
4    return (void*)currentHeap;
5 }
```

GCMRenderer localMemoryAlloc Function

What is this function doing? Well, you should be able to see that we are storing the existing pointer into graphics memory in the variable *currentHeap*, and then incrementing the pointer by a value based on the input variable *size*, and then returning the local variable. It might be easier to think of it in this way - if we imagine we have a notebook of numbered pages, and we want to reserve some pages to write something in later, we could just skip ahead that many pages, and write our next note there instead. In this example, the current page number we'll be writing to next is the current pointer position into graphics memory, and the *size* variable is how many pages to reserve for something else.

That's all well and good, but what on earth is this line actually doing?

```
1 localHeapStart      += (size + 1023) & (~1023);
```

GCMRenderer localMemoryAlloc Code Snippet

It is actually making sure that size is a multiple of 1024 bytes, to aid in data alignment. To see how it does this, let's go through a simple example. Let's assume our size is 10, and we want to make sure it is a multiple of 16, rather than 1024 (just to make the values smaller). We are going to subtract 1 from size, and then do a bitwise AND with the inverted size-1 value (you may remember the tilde bitwise operation from the stencil buffering tutorial!).

```
1 Variable        Binary          Decimal
2 size               00001010        10
3 align              00010000        16
4 align-1            00001111        15
5 ~align-1           11110000           a
6 size+(align-1)     00011001        25 b
7 a&b                00010000        16
```

GCMRenderer localMemoryAlloc byte alignment

Pretty neat! The combination of the bitwise inversion and bitwise AND on the value *align-1* will bump our answer up to the next multiple of the alignment! The last function does pretty much the same thing, but with the alignment determined by a function parameter:

```
1 void* GCMRenderer::localMemoryAlign(
2                                     const uint32_t alignment,
3                                     const uint32_t size) {
4    localHeapStart = (localHeapStart + alignment-1) & (~(alignment-1));
5    return (void*)localMemoryAlloc(size);
6 }
```

GCMRenderer localMemoryAlign Function

Ideally we would have a more complicated memory management system than this; afterall, what will happen if we try to allocate more than 256mb of RAM during a program execution? We have no way of 'deleting' memory, and so the program will fail. For simple applications this method will serve us well, though.

We have one more function call in our constructor, *InitSurfaces*. Remember, all rendering in GCM is done to a surface - the GCM equivalent to an OpenGL Frame Buffer Object. The *GCMRenderer* class has two "built in" surfaces, that we are going to use as the *front* and *back* buffer during rendering. Each surface will have its own colour buffer (as we will be rendering into one while drawing the other), but will share a single depth buffer. GCM surfaces also support multiple colour attachments - up to a total of four. You may use them in your own applications, but for our front and back buffers they are not necessary, and so we are going to disable them. Here's the start of the *InitSurfaces* function:

```
1  void GCMRenderer::InitSurfaces() {
2     depthPitch           = screenWidth*4;
3     uint32_t depthSize   = depthPitch*screenHeight;
4     void* depthBuffer    = localMemoryAlign(64, depthSize);
5     cellGcmAddressToOffset(depthBuffer, &depthOffset);
```

GCMRenderer InitSurfaces Function

First up we are going to initialise some memory to use as our depth (and as you will see shortly, also our stencil) buffer. We got a value for the local variable *screenWidth* in the *InitDisplay* function, and using it we can calculate the pitch of our depth buffer (32BPP * *screenWidth*), and the total number of bytes we will require for it (pitch * *screenHeight*). Then we can use our new localMemoryAlign function to allocate some graphics memory (the memory we use for screen buffers must be at a 64-byte alignment), and then calculate the offset of this new memory (line 5).

Next, we must do the same for the two colour buffers , and set some values in our two surfaces - it has variables to determine where the memory is (on line 17, where we set it to local), and the offset (line 12, as the second parameter). Note that the location and offset variables are actually arrays - this is for multiple colour attachments, and since we only want one colour attachment for these buffers, we must have an additional loop to disable multirendering (lines 19 - 24, with the offset and pitch signifying to GCM that these attachments are disabled). The only other interesting function here is *cellGcmSetDisplayBuffer*, which registers our surface memory as an output buffer - every time we swap buffers, GCM will move to the next registered display buffer.

```
6     uint32_t colourPitch =  screenWidth*4;
7     uint32_t colourSize  =  colourPitch*screenHeight;
8
9     for(int i = 0; i < 2; ++i) {
10
11       void*buffer = localMemoryAlign(64, colourSize);
12       cellGcmAddressToOffset(buffer, &surfaces[i].colorOffset[0]);
13
14       cellGcmSetDisplayBuffer(i, surfaces[i].colorOffset[0],
15                               colourPitch, screenWidth, screenHeight);
16
17       surfaces[i].colorLocation[0]  = CELL_GCM_LOCATION_LOCAL;
18       surfaces[i].colorPitch[0]     = colourPitch;
19       surfaces[i].colorTarget       = CELL_GCM_SURFACE_TARGET_0;
20
21       for(int j = 1; j < 4; ++j) {
22          surfaces[i].colorLocation[j]  =  CELL_GCM_LOCATION_LOCAL;
23          surfaces[i].colorOffset[j]    =  0;
24          surfaces[i].colorPitch[j]     =  64;
25       }
```

GCMRenderer InitSurfaces Function

Finally, we set some more surface parameters, including the width and height of the surface, and its depth information - note that on line 30 we use the symbolic constant CELL_GCM_SURFACE_Z24S8, which tells GCM to use 24 bits of our depth buffer for depth information, and the remaining 8 bits for stenciling.

```
26       surfaces[i].type              = CELL_GCM_SURFACE_PITCH;
27       surfaces[i].antialias         = CELL_GCM_SURFACE_CENTER_1;
28       surfaces[i].colorFormat       = CELL_GCM_SURFACE_A8R8G8B8;
29       surfaces[i].depthFormat       = CELL_GCM_SURFACE_Z24S8;
30       surfaces[i].depthLocation     = CELL_GCM_LOCATION_LOCAL;
31       surfaces[i].depthOffset       = depthOffset;
32       surfaces[i].depthPitch        = depthPitch;
```

```
33
34        surfaces[i].width              = screenWidth;
35        surfaces[i].height             = screenHeight;
36        surfaces[i].x                = 0;
37        surfaces[i].y                = 0;
38    }
39    cellGcmSetSurface(&surfaces[0]);
40 }
```

GCMRenderer InitSurfaces Function

That's the setup of GCM done! But there are a few more functions that must be defined. In OpenGL, at the start of every frame it is usual to clear the buffer being drawn into, and at the end of the frame to swap the buffers. We do exactly the same in GCM, but they require a bit more work, and so are put into their own *GCMRenderer* functions. Here's *ClearBuffer*:

```
1  void GCMRenderer::ClearBuffer() {
2
3     cellGcmSetColorMask(CELL_GCM_COLOR_MASK_R|CELL_GCM_COLOR_MASK_G|
4        CELL_GCM_COLOR_MASK_B|CELL_GCM_COLOR_MASK_A);
5
6     cellGcmSetClearColor((64<<0)|(64<<8)|(64<<16)|(255<<24));
7
8     cellGcmSetClearSurface(
9              CELL_GCM_CLEAR_Z|CELL_GCM_CLEAR_S|CELL_GCM_CLEAR_R|
10             CELL_GCM_CLEAR_G|CELL_GCM_CLEAR_B|CELL_GCM_CLEAR_A);
11 }
```

GCMRenderer ClearBuffer Function

We can think of this as our 'frame reset' function, so we enable writes into all of the colour channels using *cellGcmSetColorMask*, which behaves the same as the colour mask function in OpenGL. We can then set a clear colour, which rather than the OpenGL way of having 4 floats to determine the colour, has a single 32 bit ARGB value, so we need to do some bitshifting, eventually making the clear colour grey. Next is the actual surface clearing function, which again takes a set of values ORd together - note that we also clear the stencil (CELL_GCM_CLEAR_S) and depth buffer (CELL_GCM_CLEAR_Z) in this function, too.

Our *SwapBuffers* does much the same as OpenGL's own *SwapBuffers* function - it swaps around the buffers so that the buffer just rendered to is now displayed on screen, so that the previously displayed buffer can be drawn into instead. Just to be careful, we are first going to use the function *cellGcmGetFlipStatus* and wait if it does not return 0 - a non-zero value indicates that the hardware is currently in the process of performing a previously requested flip. Once this is done, we reset the flip status (6), flip the buffers (line 8), and then tell GCM to flush the pipeline - that is, to execute everything in its command buffer before moving on. On line 12, we invert a class member *boolean* variable *swapValue*, and then use it to inform GCM which of the two surfaces in our member variable array to render into next.

```
1  void GCMRenderer::SwapBuffers() {
2     while (cellGcmGetFlipStatus()!=0){
3        sys_timer_usleep(300);
4     }
5     cellGcmResetFlipStatus();
6     cellGcmSetFlip((uint8_t)swapValue);
7     cellGcmFlush();
8     cellGcmSetWaitFlip();
9
10    swapValue = !swapValue;
11    cellGcmSetSurface(&surfaces[swapValue]);
12 }
```

GCMRenderer SwapBuffers Function

Sometimes during graphical rendering, we might want to set the viewport transformation - that is, the transform that transforms a vertex from Normalised Device Coordinates (-1.0 to 1.0 on each axis) to screen coordinates (0 to screen width and height, plus min and max z values). You might recall that we changed the viewport during shadow mapping tutorial so that the viewport matched the dimensions of the shadow scene depth buffer. Rather than *glViewport*, which took a size and a translation, GCM has *cellGcmSetViewport*, which takes a few extra parameters. Again, setting the viewport has been wrapped up inside a *GCMRenderer* class function for you, *SetViewport*, which by default sets the viewport to fill the screen, but could easily be extended to take in parameters for situations such as shadow map rendering.

```
41 void GCMRenderer::SetViewport() {
42    uint16_t x,y,w,h;
43    float min, max;
44    float scale[4],offset[4];
45
46    x = 0;   //starting position of the viewport (left of screen)
47    y = 0;  //starting position of the viewport (top of screen)
48    w = screenWidth;  //Width of viewport
49    h = screenHeight; //Height of viewport
50    min = 0.0f; //Minimum z value
51    max = 1.0f; //Maximum z value
52    //Scale our NDC coordinates to the size of the screen
53    scale[0] = w * 0.5f;
54    scale[1] = h * -0.5f;   //Flip y axis!
55    scale[2] = (max - min) * 0.5f;
56    scale[3] = 0.0f;
57
58    //Translate from a range starting from -1 to a range starting at 0
59    offset[0] = x + scale[0];
60    offset[1] = y + h * 0.5f;
61    offset[2] = (max + min) * 0.5f;
62    offset[3] = 0.0f;
63
64    //analogous to the glViewport function...but with extra values!
65    cellGcmSetViewport(x, y, w, h, min, max, scale, offset);
66 }
```

GCMRenderer InitSurfaces Function

The vectors *scale* and *offset* can be thought of as combining to form a matrix that will scale up NDC space to the screen dimensions, and then translating the coordinates so that the origin is at the bottom left, rather than the centre of the screen.

The final GCMRenderer function we're going to look at is the SetCurrentShader function. This time, it takes two parameters - one vertex shader, and one fragment shader, but is otherwise the same as the OGLRenderer equivelent. Note that the vertex shader function requires a *pointer* to the microcode, while the fragment program function requires the *offset*.

```
1 void  GCMRenderer::SetCurrentShader(VertexShader & vert,
2       FragmentShader &frag) {
3    currentFrag = &frag;
4    currentVert = &vert;
5
6    cellGcmSetFragmentProgram(currentFrag->program,currentFrag->offset);
7    cellGcmSetVertexProgram(currentVert->program,currentVert->ucode);
8 }
```

GCMRenderer SetCurrentShader Function

Renderer Class

That's the basic base class finished with! Although the API functions were different, and we had to do a bit more work, it wasn't really *that* different than what you were used to. Just like the *OGLRenderer* base class, the *GCMRenderer* is missing a *RenderScene* function. Such a function might look like this:

```
9  void Renderer::RenderScene() {
10    SetViewport();
11    ClearBuffer();
12    this->SetCurrentShader(*currentVert,*currentFrag);
13
14    cellGcmSetDepthTestEnable(CELL_GCM_TRUE);
15    cellGcmSetDepthFunc(CELL_GCM_LESS);
16
17    modelMatrix = Matrix4::identity();
18
19    if(camera) {
20       viewMatrix = camera->BuildViewMatrix();
21    }
22    else{
23       viewMatrix = Matrix4::identity();
24    }
25
26    projMatrix  =
27    Matrix4::perspective(0.7853982, screenRatio, 1.0f, 20000.0f);
28
29    currentVert->UpdateShaderMatrices(modelMatrix,
30                                      viewMatrix,projMatrix);
31
32    if(root) {
33       DrawNode(root);
34    }
35
36    SwapBuffers();
37 }
```

GCMRenderer SetCurrentShader Function

Looks familiar, doesn't it? Although we're using a different rendering API, and different classes to support it, the end result is a renderer that works pretty much as we are used to. We can reset matrices to identity, and also generate perspective matrices (note, rather than degrees, Sony's perspective matrix function uses *radians*), send them to the vertex shader, and render our scene graph.

Cg Shaders

As described earlier, we don't use GLSL shaders when programming on the Playstation, we use Nvidia's *Cg* language. Don't get too worried though, they both use a C-style syntax, and do much the same thing, and use the same programmable model of vertex and fragment shaders. with a few minor differences. These are mostly around specific syntax, and datatypes, so let's look at an example vertex shader to see what some of these are.

On line 1, we can see a **struct** called *output* - we'll see very shortly how it is used, and how similar it is to the **out** concept in GLSL. Note that it contains two instances of a datatype **float4** - this is what Cg calls a 4-component vector. Also, each component has what is known as a *binding semantic* after it, which provides a hint to the Cg compiler what each variable is to be used for, which may help in making optimisations. On line 7, we begin our main function, and straight away we can see that rather than being a **void** type, it will return an instance of our newly described **struct**!. The main function also has 6 input parameters - our vertex attributes (complete with binding semantics), and three **uniforms** of type **float4x4** - Cg's 4 by 4 matrix datatype. If we look back at the *GCMRenderer SetDefaultAttributes* function, we'll see that the **strings** we used there match up to the input

attributes here, as do the matrices with the **strings** in *UpdateShaderMatrices*. In the main function itself, we can see that we have to declare an instance of our output struct (line 18), and at the end of our function return it (line 27). Otherwise, the function should make sense - we form an MVP matrix from the input matrices, and transform the input position by it, and additionally pass our other input parameters to the output **struct**. Note that rather than the easier multiplication operator (*), we use a multiplication function instead (mul) - the operator alternative will compile, but does not seem to provide correct answers.

```
1  struct output {
2      float4 position : POSITION;
3      float4 color    : COLOR;
4      float2 texCoord : TEXCOORD0;
5  };
6
7  output main
8  (
9      float4 position   : POSITION,
10     float4 color      : COLOR,
11     float2 texCoord   : TEXCOORD0,
12
13     uniform float4x4 modelMat,
14     uniform float4x4 viewMat,
15     uniform float4x4 projMat
16 )
17 {
18     output OUT; //We must declare our output struct
19
20     float4x4 mvp = mul(projMat, mul(viewMat, modelMat));
21     //mvp = projMat * viewMat * modelMat; //This compiles! But don't...
22
23     OUT.position    = mul(mvp, position);
24     OUT.color       = color;
25     OUT.texCoord    = texCoord;
26
27     return OUT; //and output it via return
28 }
```

vertex.cg

That's the vertex shader done with - now let's take a look at the fragment shader, which is very similar to GLSL fragment shaders. It takes in two parameters (linked to our vertex shader's output by a combination of their name and binding semantic), a uniform texture sampler, and finally an output float4 with a *COLOR* binding semantic - this is functionally identical to the output from our GLSL fragment shaders. The *samplerRECT* and *texTECT* functions are used to sample from square, power of 2 textures - we could use sampler2D and tex2D for non-square textures.

```
1  void main
2  (
3      float4 color_in      : COLOR,
4      float2 texcoord      : TEXCOORD0,
5      uniform samplerRECT texture,
6
7      out float4 color_out : COLOR
8  )
9  {
10     color_out = color_in * texRECT(texture, texcoord);
11 }
```

fragment.cg

Tutorial Summary

If everything has gone to plan, you'll have a grey screen with a multicoloured triangle in the middle of it! It seems like a lot of code to do something so simple - but like the SPU tutorial, you've learned a lot on the way, and have created a good starting point for some simple graphical rendering. You now know how to load things into the RSX processor's local memory, how to render things, how to load and use shaders, and how to swap the buffers (as well as create them!). Next lesson we'll take things a bit further, and add texture mapping to our simple renderer.

Further Work

1) Try to build a simple scene-graph based graphical scene - maybe the cube robot! Try including some shader functionality controlled by a uniform, such as an overall node colour, or a selectable blending between two texture samplers.

2) Try converting the realtime lighting functionality outlined in the graphics module to Cg and GCM - you'll need to add normals to the *Vertex* **struct**, and create some way of sending light information to to the fragment shader. Cg contains all of the vector maths required to perform this, just remember you are dealing with float3 and float4, rather than vec3 and vec4.

3) Writing to the stencil buffer can be enabled using *cellGcmSetStencilTestEnable*, *cellGcmSetStencilFunc*, and *cellGcmSetStencilOp*. Try implementing the stencil buffering tutorial