Lesson 7 - Texturing With GCM

Summary

As a current generation console, the Playstation 3 is able to utilise all of the texturing functions you are used to - mipmapping, texture filtering, and anisotropy are all features of the RSX graphics processor. However, the methods for loading and accessing textures are a bit different to what you might be used to. In this tutorial, we are going to look at how to load texture maps into the graphics memory of the Playstation 3, and how to enable the various texture filtering methods available for use.

New Concepts

GCM Texturing, Packed Texture Formats, GTF Textures, Texture Memory Layout

Tutorial Overview

In this second GCM tutorial, we're going to take a look at the texturing of primitives. Texturing should be familiar to you from the Graphics For Games module, and we can do everything we could do in that module using the graphics processor of the Playstation 3 - the Nvidia RSX. Unlike in OpenGL, where we could let the graphics card drivers do the dirty work of texture memory management, in GCM we must handle all memory allocation ourselves. As we will see shortly, this can get tricky to deal with when using certain texturing filters, such as mipmapping. Fortunately, there are some tools that will help mitigate the added complexity of dealing with texturing on the RSX somewhat.

Texturing Using The RSX And GCM

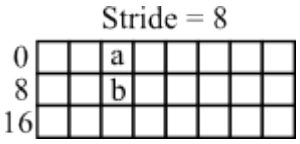
When dealing with texturing in OpenGL in the Graphics For Games module, we didn't have to think much about what texture data is, or how to handle it - we even had a library that would do the hard work for us! Most of the time, we only had to think of a texture as an OpenGL *name* - a simple **unsigned int**. It is not quite so simple in GCM - for one we must load the texture data into memory ourselves, and instead of an **unsigned int** to identify a texture, GCM uses the *CellGcmTexture* **struct**, which includes members to keep track of texture information such as the width, height, and the texture's offset into RSX memory. Other than that, texturing works much the same as you are used to, via *binding* to *samplers* and accessed via *texture coordinates*, with support for filtering, mipmapping, and anisotropy, as well as specialised texture formats such as cubemaps.

Texture Memory Considerations

As with most data we keep in graphics memory for use during rendering, we must ensure that the texture data is correctly byte *aligned*. Depending on their exact type, generally a texture must be either 64 or 128-byte aligned. Some texture types, such as cube maps, must also have dimensions that are a power of two in size, and as we will shortly see, keeping texturing to power of two sizes can increase both memory access and rendering performance.

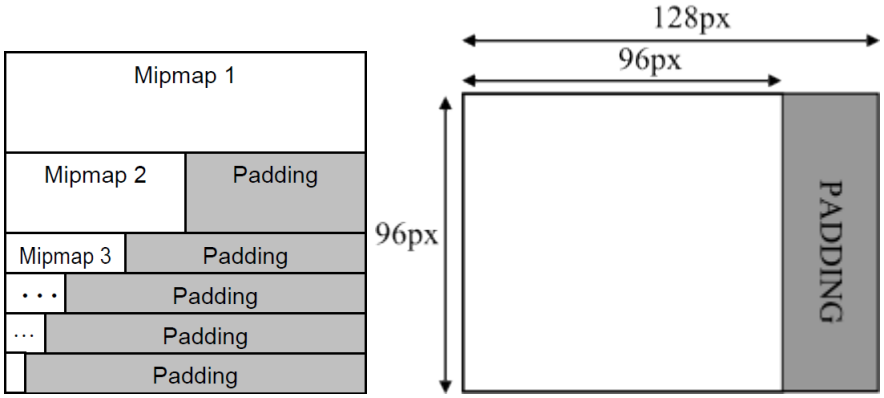
The Nvidia RSX supports mipmapping for rendering efficiency, and this should be used whenever possible in order to reduce the memory bandwidth requirements - smaller textures will lead to more correct texels being in the texture cache. Memory bandwidth is something we didn't really have to consider during the Graphics For Games module, due to the modern high-end graphics card in the available machines - but every little bit of saved bandwidth (that is, reducing how many times the GPU must access its memory to sample a texture or obtain vertex data during the rendering of a frame) will increase the performance of the application. For example, as of writing, the lab machines have a GPU memory bandwidth of 163.9GB of data transfer per second, while the RSX in the Playstation is limited to 22.4GB of data transfer per second. The texture memory itself is probably more limited than you are used to dealing with, too - while a modern gaming PC might have over a gigabyte of graphics memory, the RSX has 256MB of RAM available to it.

When dealing with some types of texture when programming on the Playstation, we must consider certain requirements for alignment with the *pitch* of a texture. The texture pitch is simply the texture's width multiplied by the number of bytes per texel (this would probably be 3 for an RGB texture, or 4 for an RGBA texture). When storing the mipmaps for a texture, each row of data must be aligned to a multiple of the source texture's pitch - so that internally, adding *pitch* to the memory address of a texel will result in being at the memory address of the texel directly below:



Adding pitch to the memory location of texel a will yield the memory address of texel b

There is a limitation when storing mipmapped textures that each row of the texture must begin on a multiple of the pitch - *even the lower-sized mipmaps*! To see how this can effect the amount of space taken up by a texture, consider the following example:

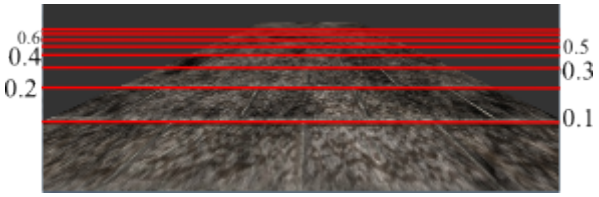


Left: Mipmap padding by stride Right: Non Power-of-2 padding by stride

As we can see, after the first mipmap level (the original texture) we end up having to have a lot of 'padded' space when storing a mipmapped texture - space we can't otherwise use due to its interleaving with texture data, but which nonetheless takes up space in our already limited 256MB of VRAM! This padding requirement can even hit us for non mipmapped textures - if the width of a texture is not a *power of two*, there must be padding bytes at the end of each row of texture data bringing the row up to a power of two in size. Another reason to limit ourselves to power of two texture sizes!

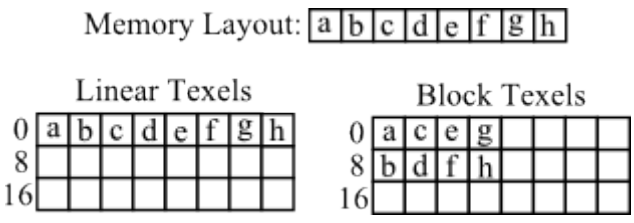
Fortunately, there is a way of avoiding the padded bytes texturing issue, and improve rendering performance too - *texture swizzling*. You may remember the term swizzling from dealing with vectors in shaders; there swizzling described the ability to access vector elements in a non-linear order. It means much the same in texturing - instead of storing a texture in memory in a linear fashion (that is, each row of the texture is stored consecutively, in a left-to-right fashion), a swizzled texture stores each individual texel in an ordering that is more matched to how the graphics hardware accesses it while rendering.

What is this ordering, and how can it improve rendering performance? To answer this, we must take a step back to graphics rendering theory, and consider what texture information is required when texturing using mipmaps. As you should remember from the Graphics For Games module, mipmapping is used to reduce the number of texture accesses the hardware must perform, by sampling lower resolution copies of a source texture in cases where there are more texels in the source image than there are fragments to display them - known as *minification*. This typically occurs when a object is in the distance of a scene, but can also occur when very large textures, or when large amounts of texture wrapping is occurring. To calculate when this is the case, the rasterising hardware will typically examine the screen space *partial derivative* of the texture coordinates being used to sample a texture - the higher this value, the more a texture coordinate is changing across the primitive's fragments, and so therefore the greater chance of skipping texels, resulting in an incorrect fragment colour unless mipmapping is used. The easiest way to calculate this derivative is to simply use the interpolated texture coordinates of adjacent fragments, and calculating the rate of change on each axis. For this reason, when fragments are being processed by the fragment shader, they are usually processed in groups of 4 adjacent fragments at a time - the derivatives can then be calculated, and used to calculate the correct mipmap and amount of anisotropy to apply.



As the textured quad projects into the screen, the screen space rate of change on the y axis changes, which can be determined by examining the rate of change of adjacent pixels

If the fragment shader is always performing texturing in a 2*2 block of fragments at a time, it is quite likely that multiple rows of the texture will need to be sampled - and this will definitely be the case when bilinear filtering is used. But if our texture is stored in a linear fashion, then we will never get access to that second row 'for free' by via loading texture data into the texture cache - there will be regular and frequent 'cache misses'. From a rendering point of view, wouldn't it be much better to store texture data in 2*2 blocks, rather than linearly? That's what swizzling is! Instead of linearly, swizzled textures are stored in blocks of 4 at a time, increasing the chance of the correct texel being in texture cache when required.



An example of how texels are arranged in memory when textures are saved linearly (left) and swizzled (right)

GTF Textures

As we're going to have to manually deal with all loading of textures ourselves, it is probably quite useful to think about how to load an image file containing a texture into graphics memory. On the PC, the graphics driver almost certainly internally stores the textures you upload in a swizzled format suitable for your graphics hardware, but since we have to handle graphics memory ourselves, we can't rely on anything to swizzle our texture data, and most easy to load picture formats are in a linear format.

Fortunately, Sony have created a special graphics file container format, *GTF*. This container format can store multiple textures in a single file, including all the sides of a cubemap, and entire mip-map chains - and what's better, is that if the texture is a power of two in size, the texture data inside the file will be in a swizzled format ideal for loading directly into texture memory! So as long as we keep to power-of-two textures, and save textures in the GTF format, we don't really have to worry too much about wasted memory from padding mipmaps and texture row widths - **phew**! Even better, the RSX can make use of *compressed* textures, which is supported by the GTF format, allowing us to save even more space (at a slight performance and fidelity hit). And it gets better! GTF files also have a *CellGcmTexture* **struct** embedded in them, so we don't even have to make our own! For these reasons, GTF should really be your primary texture file format on the PS3 in order to maximise performance.

So how do we create a GTF texture? It is relatively straightforward, but does require the use of an external program. The GTF file format is quite similar to another container format, Microsoft's DDS file, and the Playstation SDK comes with a tool (it's simply called DDS2GTF.exe) that will convert a DDS file to GTF, simply by dropping a DDS file onto it. Many popular image editing programs can save as DDS, including paint.net, so it's easy to convert any image we need into a GTF file, simply by exporting it as a DDS using paint.net, and then turning it into a GTF file using DDS2GTF.

Example Code

To complete our look at how texturing is performed on the Playstation, we're going to take a look at some of the functions in the basic Playstation 3 framework that accompanies this tutorial series. In doing so, we'll learn how to create and fill a *cellGcmTexture* struct, and how to load some texture data into the Playstation's graphics memory. To use this texture data, we'll also look at the *GenerateQuad* and *Draw* functions in the Mesh and *GCMRenderer* classes, respectively.

GCM Renderer *SetTextureSampler* Function

As you should recall from working with OpenGL, there are two parts to accessing a texture in a shader - *binding* the texture to the texturing hardware, and *setting* the shader uniform *sampler* with the appropriate value. Due to how closely tied with the actual RSX graphics hardware the GCM API is, these two processes are effectively combined into a single *set* function call. However, while in OpenGL we could happily set texturing parameters such as coordinate clamping and filtering settings once per texture, in GCM these settings are considered to be part of the texturing hardware, and are thus set on a per texture *unit* basis, rather than per texture. One final thing to consider is how susceptible GCM and the RSX are to incorrect parameters. Most OpenGL implementations will 'behave' themselves if given incorrect parameters, and will at worst render nothing, or black fragments. GCM however, is very 'stripped back' to maintain performance, and expects the programmer to identify any incorrect parameters before they are sent to the API.

Due to these considerations, the GCM rendering framework you are provided with comes with a function that will correctly bind a texture to a shader uniform - *SetTextureSampler*.It takes in two parameters - a *CGparameter*, and a pointer to a *CellGcmTexture*; We can think of these as equivelent to OpenGL's uniform index, and texture name, respectively. On line 3 of the function, there's a check to make sure the sampler parameter is not 0 - much as OpenGL will return a specific 'not found' value, requests for non-existant uniforms will return 0 in GCM. Unlike OpenGL however, trying to set data on a non-existant uniform will almost certainly crash the program, so we must be careful to avoid this by **returning** if necessary.

On line 6, we have a new datatype (*CGresource*), and a new function *cellGcmCgGetParameterResource*. This obtains the mapping between the uniform (the parameter) and the actual RSX hardware (the resource); essentially, this determines which hardware texture unit the shader uniform will be bound to. Then, on line 10, we check whether the texture we want to bind actually exists or not, and if not, turns off the texture unit and **returns**. If all goes well, the function proceeds to line 15, where the texture is bound to the actual RSX texture unit hardware, which is then switched on on line 17.

```
1  void  GCMRenderer::SetTextureSampler(CGparameter sampler,
2                                       const CellGcmTexture *texture) {
3     if(!sampler) {
4        return;
5     }
6     CGresource unitResource = (CGresource)
7     (cellGcmCgGetParameterResource(currentFrag->program, sampler)
8     - CG_TEXUNIT0);
9
10    if(!texture) {
11       cellGcmSetTextureControl(unitResource, CELL_GCM_FALSE, 0, 0, 0);
12       return;
13    }
14
15    cellGcmSetTexture(unitResource, texture);
16
17    cellGcmSetTextureControl(unitResource, CELL_GCM_TRUE, 0, 0, 0);
```

GCMRenderer SetTextureSampler Function

Finally, this function sets the texture coordinate and comparison state of the hardware texture unit, using the cellGcmSetTextureAddress function. This single function is used for many differing types of textures, and so has some parameters that won't be necessary for standard 2D textures. For example, the 2nd, 3rd, and 4th parameters determine whether texture clamping or texture repeating are used for each axis of the texture - the RSX hardware supports 3D textures, and the parameter must be set even for 2D textures. Parameter 5 allows texture coordinates used on this texture unit to be remapped from the range 0.0 - 1.0 to -1.0 - 1.0, while the final parameter is used for shadow map comparisons.

The final GCM call in this function is to *cellGcmSetTextureFilter*, which sets the filtering method - as with OpenGL, we can set nearest neighbour, bilinear, and mipmapping filtering types, for minification and magnification. The final parameter to *cellGcmSetTextureFilter* takes a bit of explaining - the RSX hardware has support for special texture filtering kernals for use when *antialiasing* is enabled - a series of techniques to disable the jagged edges around polygons, usually involving generating multiple screen samples in the rasteriser. These special filtering kernals take more texture samples than a normal bilinear filter, which when combined with antialiasing methods will help hide the polygon edges, at the cost of a slightly blurrier final image. By default this framework doesn't perform any antialiasing, so this field is left at a valid default value - quincunx filtering, which takes 5 texture samples in the shape of an x (perhaps better visualised as the five dots on the 5th side of a die). To enable this advanced texture filtering, the texture filtering type should be set to the **CELL_GCM_TEXTURE_CONVOLUTION_MIN** symbolic constant.

```
18    cellGcmSetTextureAddress(unitResource,
19       CELL_GCM_TEXTURE_CLAMP,
20       CELL_GCM_TEXTURE_CLAMP,
21       CELL_GCM_TEXTURE_CLAMP,
22       CELL_GCM_TEXTURE_UNSIGNED_REMAP_NORMAL,
23       CELL_GCM_TEXTURE_ZFUNC_LESS,
24       0);
25
26    cellGcmSetTextureFilter(unitResource, 0,
27       CELL_GCM_TEXTURE_LINEAR,
28       CELL_GCM_TEXTURE_LINEAR,
29       CELL_GCM_TEXTURE_CONVOLUTION_QUINCUNX);
30 }
```

GCMRenderer SetTextureSampler Function

Here's an example of how to call the *SetTextureSampler* function, using the *GCMRenderer* equivalent of the *DrawNode* function introduced in the Graphics For Games module. We are asking the fragment shader for the *CGparameter* called 'texture', and asking a *SceneNode's* mesh for it's default texture. Due to the safety checks in the *SetTextureSampler* itself, this function call will not cause a crash even if the *Mesh* has no texture, and the current fragment shader has no *texture* **uniform** parameter.

```
1  void  GCMRenderer::DrawNode(SceneNode*n)  {
2     if(n->GetMesh()) {
3        Matrix4 transform = n->GetWorldTransform();
4        currentVert->SetParameter("modelMat", transform);
5
6        SetTextureSampler(currentFrag->GetParameter("texture"),
7                          n->GetMesh()->GetDefaultTexture());
8
9        n->GetMesh()->Draw(*currentVert,*currentFrag);
10    }
11 }
```

GCMRenderer SetTextureSampler Function Usage Example

GCM Renderer *LoadGTF* Function

The GTF texture file format should be your primary texture format when creating graphical scenes on the Playstation. Fortunately, loading a GTF texture is pretty easy, and in the process of doing so, we'll get a valid *CellGcmTexture* **struct** for the new texture data, too!

In the provided framework, the *LoadGTF* function takes care of loading a texture from a file, and returning a *CellGcmTexture* struct for use elsewhere in the program. The first few lines of the program simply create an **ifstream** for the filename - remember, we can use standard C++ constructs on the Playstation! Note that as with any other file operation, we append the filename to the macro **SYS_APP_HOME**, which will be set by the Visual Studio solution properties to the correct folder.

The GTF file format is pretty simple to load in when using the GCM provided structs. At the start of the file is a *CellGtfFileHeader* **struct** - from the data inside this we can determine how many textures are inside the GTF container. So, on line 14 we can define a local variable of the *CellGtfFileHeader* type, and then on line 15 read its contents straight from the file, using *ifstream::read* and the **sizeof** operator. Inside the newly read **struct** there's a *NumTexture* parameter - we can use this to initialise an array of *CellGtfTextureAttribute* **structs** - there's one of these for each texture inside the GTF, in a linear chunk of data after the initial *CellGtfFileHeader*. Due to how these **structs** are tightly packed in the file, we can load them all in at once using a single read, passing the read function a **pointer** to the start of the first attribute **struct**, and multiplying the **sizeof** by the number of **structs**.

```
1  CellGcmTexture* GCMRenderer::LoadGTF(std::string filename) {
2     CellGcmTexture* texture = new CellGcmTexture();
3
4     std::ifstream file;
5
6     std::string folder = SYS_APP_HOME + filename;
7
8     file.open(folder.c_str(), std::ios::binary);
9     if(!file.is_open()){//File hasn't loaded!
10       delete texture;
11       return NULL;
12    }
13
14    CellGtfFileHeader header;
15    file.read((char*)&header,sizeof(CellGtfFileHeader));
16
17    CellGtfTextureAttribute*attributes =
18          new CellGtfTextureAttribute[header.NumTexture];
19
20    file.read((char*)attributes,header.NumTexture *
21          sizeof(CellGtfTextureAttribute));
```

GCMRenderer LoadGTF Function

For now, we will assume there is only a single texture in the GTF file; this will usually be the case, as even the six sides of a cube map are considered a single combined texture. As each *CellGtfTextureAttribute* struct comes with its own *CellGcmTexture* struct already filled in, we can simply do another **memcpy** (on line 24) to copy the details into the memory we're going to be returning a pointer to. The *CellGtfTextureAttribute* also has an *OffsetToTex* member variable, which tells us where the start of the actual GTF texture data is in the file, and a *TextureSize* variable which tells us how much data there is. Using these, we can seek through the file to that position (line 27), intialise some (correctly byte aligned!) graphics memory (line 29), read the texture data into the newly initialised memory (line 31), and finally use *cellGcmAddressToOffset* (line 33) to determine the correct offset value, for the GCM functions that require it.

```
23    for(int i = 0; i < min(1,header.NumTexture); ++i) {
24       memcpy((void*)texture,(void*)&(attributes[i].tex),
25             sizeof(CellGcmTexture));
26
27       file.seekg(attributes[i].OffsetToTex);
28
29       char*rsxdata = (char*)localMemoryAlign(128,
30                      attributes[i].TextureSize);
31       file.read(rsxdata,attributes[i].TextureSize);
32
33       cellGcmAddressToOffset( rsxdata, &texture->offset );
34    }
35
36    delete[] attributes;
37
38    return texture;
39 }
```

GCMRenderer LoadGTF Function

Once this has been done, we no longer require the attributes data, and so can **delete** it, and finally **return** the texture variable **pointer** - which has now been filled with the relevent data to access the texture by the **memcpy** on line 24.

Mesh Class *GenerateQuad* Function

For the next code snippet in this tutorial, let's take a look at how to actually create a mesh with texture coordinates, using the PS3 version of the *GenerateQuad* function as an example. It begins much as the *GenerateTriangle* function described in the previous function - we create a new *Mesh* (line 2), define its vertex and index count (line 4-5), and allocate the correct amount of GPU memory (lines 7 and 18).

```
1  Mesh* Mesh::GenerateQuad() {
2     Mesh*m = new Mesh();
3
4     m->numIndices    = 6;
5     m->numVertices    = 4;
6
7     short *indices =
8        (short*)GCMRenderer::localMemoryAlign(128, sizeof(short) * 6);
9
10    indices[0] = 0;
11    indices[1] = 1;
12    indices[2] = 2;
13
14    indices[3] = 2;
15    indices[4] = 3;
16    indices[5] = 0;
17
18    m->vertexData =
19       (Vertex*)GCMRenderer::localMemoryAlign(128, sizeof(Vertex) * 4);
```

Mesh Class GenerateQuad Function

During the OpenGL programming tutorials, we were creating separate vertex streams for each vertex component - one for positions, one for texture coordinates etc. Here, we are instead creating a single *interleaved* vertex buffer, with all of the vertex components for a single vertex packed together. We do so by creating an **array** of the *Vertex* **struct**, and filling its *xyz* components with position

data, and *uv* components with texture coordinate data - this could be extended to supporting normals and tangents simply by adding more **floats** to the *Vertex* **struct** (*nx,ny,nz* and *tx,ty,tz* perhaps).

```
20    float size = 1.0f;
21
22    m->vertexData[0].x = -size;   //Top left of our quad
23    m->vertexData[0].y = size;
24    m->vertexData[0].z = 0.0f;
25    m->vertexData[0].u = 0.0f;
26    m->vertexData[0].v = 1.0f;
27
28    m->vertexData[1].x = -size;      //Bottom left
29    m->vertexData[1].y = -size;
30    m->vertexData[1].z = 0.0f;
31    m->vertexData[1].u = 0.0f;
32    m->vertexData[1].v = 0.0f;
33
34    m->vertexData[2].x = size;       //Bottom Right
35    m->vertexData[2].y = -size;
36    m->vertexData[2].z = 0.0f;
37    m->vertexData[2].u = 1.0f;
38    m->vertexData[2].v = 0.0f;
39
40    m->vertexData[3].x = size;       //Top Right
41    m->vertexData[3].y = size;
42    m->vertexData[3].z = 0.0f;
43    m->vertexData[3].u = 1.0f;
44    m->vertexData[3].v = 1.0f;
45
46    m->vertexData[0].rgba=0xffffffff;   //It'll be all white!
47    m->vertexData[1].rgba=0xffffffff;
48    m->vertexData[2].rgba=0xffffffff;
49    m->vertexData[3].rgba=0xffffffff;
```

Mesh Class GenerateQuad Function

As we are using an interleaved VBO, we must set the offset to each vertex component, so that the RSX processor can stride over vertex components correctly, just as was demonstrated in the previous tutorial:

```
50    cellGcmAddressToOffset( &m->vertexData->x,
51                            &m->vertexOffsets[VERTEX_POSITION]);
52    cellGcmAddressToOffset( &m->vertexData->rgba,
53                            &m->vertexOffsets[VERTEX_COLOUR]);
54    cellGcmAddressToOffset( &m->vertexData->u,
55                            &m->vertexOffsets[VERTEX_TEXCOORD]);
56    cellGcmAddressToOffset( indices, &m->vertexOffsets[VERTEX_INDEX]);
57
58    return m;
59 }
```

Mesh Class GenerateQuad Function

Mesh Class *Draw* Function

When drawing a Mesh using GCM, we should set the vertex data array for the texture coordinates in the same way as we did for vertex position information in the previous tutorial - we use *cellGcmSetVertexDataArray* to define the size (line 5) and format (lines 6-8) of the vertex data, give it an offset into the vertex struct to use as a stride (line 9).

```
1    if(vertexOffsets[VERTEX_TEXCOORD])  {
2       cellGcmSetVertexDataArray(
3          vertex.GetAttributeIndex(VERTEX_TEXCOORD),
4          0,
5          sizeof(Vertex),
6          2,
7          CELL_GCM_VERTEX_F,
8          CELL_GCM_LOCATION_LOCAL,
9          (uint32_t)vertexOffsets[VERTEX_TEXCOORD]
10      );
11   }
12   else{
13      cellGcmSetVertexDataArray(
14         vertex.GetAttributeIndex(VERTEX_TEXCOORD),0,0,0,
15         CELL_GCM_VERTEX_F,CELL_GCM_LOCATION_LOCAL,0
16      );
17   }
```

Mesh Class Draw Function Snippet

Tutorial Summary

You should now know the basics of how to load and use textures in your Playstation 3 programs. From here you should be able to begin to apply the texturing techniques introduced during the course of the *Graphics For Games* module - cube mapping, rendering to a texture, post processing, and anisotropic filtering are all available for you to use, although you may need to examine the SDK documentation and code samples to see how to apply them.

Further Work

1) Try rendering into a texture! You will need a screen-sized area of memory, a new *CellGCMSurface* (think of these as being your FBO), and a new *CellGCMTexture*.

2) Try loading in and sampling from a cube map. The Cg datatype for a cube map sampler is **samplerCUBE**. The 6 sides of the cubemap should be together as a single chunk of memory, with the caveat that each face must lie on a 128-byte boundary (or you may wish to find a DDS file format cubemap to convert directly to GTF to make this easier).

3) Try performing shadow mapping in your scene - the theory is the same no matter which API you use! As with further work 2, you will need some graphics memory that does 'double duty' as both a depth pointer for a *CellGcmSurface*, and as the memory for a *CellGCMTexture*. What data will need to be set in the *CellGCMTexture* struct of the shadow map?