Lesson 5
GCM Graphics Framework

Playstation 3 Development

Sam Serrels and Benjamin Kenwright*

Abstract
This article explains to the reader how to create a reusable graphics framework for rendering multiple 3D objects using independent shaders. The main rendering code is very similar to the code in the “Introduction to GCM” tutorial, now it will be split into classes and expanded upon. A much larger and fully featured game engine could be built up from the code created here, using it as a well structured base to iterate upon.

Keywords
Sony, Graphics, Shaders, PS3, PlayStation, Setup, GCM, Target Manager, ELF, PPU, SPU, Programming, ProDG, Visual Studio

* *Edinburgh Napier University, School of Computer Science, United Kingdom*: b.kenwright@napier.ac.uk

Contents

Introduction

About the Edinburgh Napier University Game Technology Playstation 3 Development Lessons Edinburgh Napier University Game Technology Lab is one of the leading game teaching and research groups in the UK - offering students cutting edge facilities that include Sony’s commercial development kits. Furthermore, within the Edinburgh Napier Game Technology group are experienced developers to assist those students aspiring to releasing their own games for PlayStation. Student have constant access to he Sony DevKits and encourage enthusiastic students to design and build their own games and applications during their spare time [1].

This Tutorial In the previous GCM tutorial, everything was contained in one file, and in one single main function. In this tutorial we will split the code up into classes and add in some extra functionality. This code will be a foundation that will enable us to build up from in later tutorials. Essentially this is the start of what would be a game engine, but at the moment it only does what the previous code did, but in a more organised and maintainable manor.

Engine optimisation It is tempting to think of an ’Ultimate Game Engine’, where everything is written once and just works from then on, it supports multiple platforms, and you can keep releasing game after game on it without rewriting any of the lower levels. This system would be a ’perfect’ system from a software design standpoint, but in reality, even if a games engine did manage to have all these features, it would probably run terribly on a PS3. A games console like the Playstation3 requires very specific optimisations to get the most out of it’s hardware. This is the same for every platform, and most of the time, it’s not economical to chase after the last 10fps in a system, however with the PS3 being completely multi-core orientated and with limited raw graphics horsepower, that last 10fps means much more if you game is currently only running at 20fps in the first place.

Code reusability vs Code performance So we have established a generic ’all sizes fit one’ engine approach just wouldn’t be efficient on a games console. The lowest levels of an engine must be orientated with the ethos of each system, in the case of the PS3 the SPUs must be *efficiently* utilised to get the most performance possible. A game engine could still be written for

multiple platforms, with the low-level back-end swapped out for a platform specific optimised version. The front-end of the engine could stay the same across all systems, with features like a file system wrapper to abstract the loading of files. But on a PS3, even loading files from disk is an involved task. So the challenge is to find the line between 'easy to use and maintain code' and 'fast, optimised platform code' and how to bridge between them effectively. This is the challenge of true multi-platform code, and it is why large teams spend years building and rebuilding engines.

1. Framework architecture

The code we will be writing (or reorganising) will accomplish the same thing as the previous code: initialise the memory, screen, buffers, load shaders, create geometry, render. This new code will do it in a more robust and extensible manor, it allows us to have multiple shaders and multiple sets of mesh data. This allows for rendering of more than one object, which can be any combination of mesh data and shaders.

3D The new code will operate in 3D space rather than 2D, this means we must enable depth buffering and a host of other render flags. Thanks to the modern rendering pipeline, this is a trivial task and results in not much more code needed than with setting up 2D rendering. The Shades will need a Model View Projection matrix to accommodate for 3D data, so we will write to generate these and send them to the Shader. With the matrix libraries available on the ps3 system, and with the robust CG shader framework, this also works out to be quite a simple task. The real challenges that this tutorial addresses are data management and design, the act of managing all the data and shaders in the best possible way.

Loading shaders In the previous GCM tutorial, shaders were hard coded into the program. In this tutorial, the shader files will be loaded by the application from a file. The code to do the loading is covered in the GCMShader code, listing:6. The system needs the shaders to already be compiled, something we have to with command lien tools. Wouldn't it be good if we could set-up visual studio to compile the shaders for us during the build process, so we can edit the shaders as much as we like? As it so happens, we can!

Automatically Compiling shaders We will use a custom build command to compile the shaders. This can be done via the visual studio interface, but for the exact features we need, it's easier to edit the project file manually. Firstly, add the shader files (fs_basic.cg and vs_basic.cg) to you project, save and close visual studio. Find the project (.vcxproj) file and open it in a text editor. Find the following in the file:

```
<ItemGroup>
  <None Include="fs_basic.cg" />
  <None Include="vs_basic.cg" />
</ItemGroup>
```

And replace with listing:1.

Listing 1. project.vcxproj custom build step

```
<ItemGroup>
  <CustomBuild Include="fs_basic.cg">
    <FileType>Document</FileType>

    <Command
    Condition="'$(Configuration)|$(Platform)'=='Debug|PS3'">
    $(SCE_PS3_ROOT)\host-win32\Cg\bin\sce-cgc -quiet
    -profile sce_fp_rsx -o "$(OutDir)/%(Filename).fpo"
    "%(FullPath)" </Command>

    <Outputs Condition=
    "'$(Configuration)|$(Platform)'=='Debug|PS3'">
    $(OutDir)/%(Filename).fpo;%(Outputs) </Outputs>

  </CustomBuild>
  <CustomBuild Include="vs_basic.cg">
    <FileType>Document</FileType>
    <Command
    Condition="'$(Configuration)|$(Platform)'=='Debug|PS3'">
    $(SCE_PS3_ROOT)\host-win32\Cg\bin\sce-cgc -quiet
    -profile sce_vp_rsx -o "$(OutDir)/%(Filename).vpo"
    "%(FullPath)"</Command>

    <Outputs Condition=
    "'$(Configuration)|$(Platform)'=='Debug|PS3'">
    $(OutDir)/%(Filename).vpo;%(Outputs)</Outputs>

  </CustomBuild>
</ItemGroup>
```

This will output the compiled shaders into the output"$(OutDir)" directory, where the .elf file should be. Depending on our project set-up you may need to change some directory settings. Make sure that the target directory for debugging/ the file serving directory contains the compiled shaders.

1.1 Code Design: Shaders

With a large code redesign task, the question is where to start. The Shader seems to be a good a place as any, we will have two types: Vertex and Fragment, each has slightly different characteristics and many similarities. This seems like an obvious candidate to be a class strucutre. An abstract *GCM_Shader* parent class, with a *GCM_VertexShader* and *GCM_FragmentShader* inheriting from it.

Shader Methods The functions our Shader class needs to have is as follows:

LoadCompiledShader(filename);
Loads a compiled shader program from the filesystem.

Initialise()
Do required setup, e.g send code to RSX local memory

SetPrameter(pramerter, value)
Set a shader parameter to the supplied value

The fragment and vertex classes will override these with their own specific implementations.

1.2 Code Design: Rendering / platform

We will need a class that initialises the GCM library, the screen and the buffers. As this class will hold all the required information, we should probably use it to contain rendering functions like clearing and swapping buffers, and setting up the viewport each frame. We could split this into two classes, a *renderer* class and a *platform* class, but in the interest of keeping it simple we

will just stick with one.

GCM_Renderer In a larger system, the *GCM_Renderer* would be inherited from a class called something like *Renderer*, which would have a version for each render framework (E.g. GCM, PSGL, OGL, DX3D). This is where things would get tricky as you would have to make large decisions on where to use abstract inheritance and where to use #Ifdefs to split up the code. Thankfully, this is an issue that we don't need to worry about right now, but you should keep this in mind.

Renderer Methods Some of the functions our Renderer class needs to have is as follows:

InitDisplay()
Set the resolution and output formats of the display

InitSurfaces()
Create the buffers in memory to render onto

SetViewport()
Setup the coordinate scaling and viewport settings

setupFrame()
Set the appropriate render flags

clearSurface()
Clear a buffer

swapBuffers()
Display the current buffer and set the alternate buffer as the render target.

1.3 Code Design: Mesh Data

Objects that we render are made up from a collection of vertices and other data. The fundamental underpinning of this section of our code is that we can have multiple object on-screen with different transformations, but they can all use the same set of mesh data. If we create the mesh data for a sphere, and store it in memory, we can have multiple spheres being rendered at different positions, without copying the mesh data, we just need to save the individual *transformations*.

Mesh Storage With the need to only store one mesh per shape, you may be thinking of creating some sort of storage class to handle which meshes are loaded. This would be necessary in any system larger than the one we are creating, but in this tutorial we will only be using one set of mesh data so we won't need a storage manager just yet

Mesh Structure

bool isloaded
Flag to determine if the mesh has been loaded into VRAM

vertexData
The vertex data, residing in main memory

GCM_FragmentShader* fragShader
The fragment shader to use for this mesh

GCM_VertexShade * vertShader
The vertex shader to use for this mesh

stVertex* vertexBuffer
Pointer to the vertex data in VRAM, when loaded

1.4 Code Design: Rendering Meshes

The data within a mesh structure could vary e.g, in the future you will almost certain want to include indexed vertices. We need a class to take in a Mesh and render it, this could be done within *GCM_Renderer*, it's name would imply that it does rendering, however we will actually be writing a new class.

GCM_MeshManager this will deal with the rendering and loading of meshes. Another design pattern would be to have Mesh as a class, and these function within it, rather than a structure with a separate manager. This desiccation based on how much data you have within a mesh and how much functionality you need it to have. For now, a simple structure suits our needs.

GCM_MeshManager function The purpose of this class is to bridge between the shaders, rendering code, and mesh data. When it needs to render a mesh it will send the data to the RSX, activate the required shader, set the appropriate render flags (e.g striped, indexed..) and then render the object

MeshManager Methods

render(mesh, mvp)
Render a Mesh object with the supplied model view Projection matrix

loadOnGPU(mesh)
Load a mesh object onto the RSX local memory, in prep for rendering

In the future, you could have functions in here that deal with parsing .obj files.

1.5 Code Design: Main

Finally, we need code to tie all the components together and unlike the previous tutorial, we will try to keep the *main.cpp* as simple as possible. This marks the first layer of abstraction that we have brought into our system. The main file still needs some PS3 libraries, and needs to know what platform is being used. In future systems you would want to abstract further up from here, to have something like a *game* class that doesn't need ot know anything about the platform it's running on.

2. Framework Code

2.1 utility classes

2.1.1 asserts

Listing 2. asserts.h

```
1 #pragma once
2 #include <stdio.h> //Printf, puts
3 #include <stdlib.h> //abort, exit
4
5 //#define HALT { std::abort(); } //abort prefered over exit.
6 //#define HALT { exit (1); }
7 #define HALT { __asm__ volatile( "trap" ); }
```

```
8
9 // call DBG_HALT on assert fail
10 #define ASSERT(exp) { if ( !(exp) ) {HALT;} }
11
12 // Prints the suplied string on assert fail, then call DBG_HALT
13 #define ASSERT_M(exp,msg) { if ( !(exp) ) {puts (msg); HALT;} }
14
15 // Calls the suplied function on assert fail, then call DBG_HALT
16 #define ASSERT_F(exp,func) { if ( !(exp) ) {func; HALT;} }
```

2.1.2 Vertex

Listing 3. common.h

```
1 #pragma once
2
3 //! Standard vertex structure with colour
4 struct stVertex
5 {
6   float x, y, z;
7   unsigned int rgba;
8 };
```

2.1.3 Mesh

Listing 4. Mesh.h

```
1 #pragma once
2 #include "Common.h"
3 #include "GCM_FragmentShader.h"
4 #include "GCM_VertexShader.h"
5 #include <vector>
6
7 struct stMesh{
8   //! Flag to determine if the data is loaded in main memory
9   bool loadedMain;
10
11  //! Flag to determine if the data is loaded on the gpu
12  bool loadedLocal;
13
14  //! Number of verticies, shortcut for vertexData.size
15  int numVerts;
16
17  //! Flag for if the verticies are in a stiped layout or not.
18  bool strip;
19
20  //! Mesh data in main memory
21  std::vector<stVertex> vertexData;
22
23  //! A reference to the Fragment shader to use
24  GCM_FragmentShader * fragShader;
25
26  //! A reference to the Vertex shader to use
27  GCM_VertexShader * vertShader;
28
29  //! Pointer to the vertex buffer in local memory
30  stVertex* vertexBuffer;
31  unsigned int vertexBufferOffset;
32 };
```

2.2 GCM Shader

2.2.1 GCMShader.h

Listing 5. GCM_Shader.h

```
1 #pragma once
2 #include <cell/gcm.h>   //for CGprogram
3 #include <vectormath/cpp/vectormath_aos.h>
4 #define Matrix4 Vectormath::Aos::Matrix4
5
6 class GCM_Shader{
7   private:
8     //! Loads a binary file form the filesytem into a char array.
9     char * LoadBinaryFile(const char* name);
10    //This would be put in a FileIO class in a larger system
11  public:
12
13    //! Loads a compiled shader from a file
14    void LoadBinaryShader(const char* name);
15
16    //! Initializes the shader program.
17    void initProgram(bool storeOnRSX);
18
19    //! Get a shader parameter by name
20    CGparameter GetParameter(const char* name);
21
22    //! Set a named paramter value
23    virtual void SetParameter (CGparameter param, float* data) = 0;
24
25    //! Set a named paramter value
26    void SetParameter (const char* name , float * data );
27
28    //! Sets the parameter as a matrix transposed to a 2d float[]
29    void SetParameterM (const char* name, Matrix4 & totranpose );
30
31    //! Sets the parameter as a matrix transposed to a 2d float[]
32    void SetParameterM(CGparameter param,Matrix4& totranpose)
33
34    //! The shader program, residing in main memory
35    CGprogram program;
36
37    //! The shader program code
38    void* ucode;
39
40    //! The memory address offset of the program
41    std::uint32_t offset;
42
43    virtual ~GCM_Shader(){};
44 };
```

2.2.2 GCMShader.cpp

Listing 6. GCM_Shader.cpp

```
1 #include "GCM_Shader.h"
2 #include "asserts.h"
3 #include "GCM_Renderer.h"
4 #include <fstream>   // std::ifstream
5 #include <string> // Memcopy
6 #include <stdio.h>   //Printf
7 #include <stdlib.h> // Malloc
8
9 // Loads a compiled shader from a file
10 void GCM_Shader :: LoadBinaryShader (const char* name )
11 {
12   //attempt to Load File
13   std :: ifstream file (name, std::ios::in|std::ios::binary );
14   char * data = LoadBinaryFile(name);
15   program = ( CGprogram )( void *) data ;
16 }
17
18 // Initializes the shader program.
19 void GCM_Shader::initProgram (bool storeOnRSX)
20 {
21   //Initialize the Cg binary program on memory for use by RSX.
22   cellGcmCgInitProgram ( program );
23
24   unsigned int ucodeSize ;
25
26   //Stores pointer to the microcode in *pUCode,
27   // and the size of the microcode into *pUCodeSize,
28   cellGcmCgGetUCode ( program , & ucode , & ucodeSize );
29
30   if(storeOnRSX)
31   {
32     //Reserve some local memory to store the shader microcode
33     void* RSXfragAddr =
34         GCM_Renderer::localMemoryAlign(64, ucodeSize);
35
36     //Copy the microcode into RSX local memory
37     memcpy (RSXfragAddr, ucode, ucodeSize);
```

```
38
39      //Convert a local memory address into an offset value.
40      //If successful, the offset value will be stored in &offset.
41      cellGcmAddressToOffset (RSXfragAddr, & offset );
42    }
43
44    printf("Shader loaded\t Size: %i bytes\n", ucodeSize);
45 }
46
47 // Get a shader parameter by name
48 CGparameter GCM_Shader::GetParameter(const char* name)
49 {
50    CGparameter p =cellGcmCgGetNamedParameter(program, name)
51    ASSERT_F(p,printf(" Can't find named parameter: %s\n",name));
52    return p;
53 }
54
55 // Sets the parameter as a matrix transposed to a 2d float[]
56 void GCM_Shader::SetParameterM(const char* p, Matrix4& mat)
57 {
58    Matrix4 tempMatrix = transpose ( mat );
59    SetParameter (p , ( float *)& tempMatrix );
60 }
61
62 // Sets the parameter as a matrix transposed to a 2d float[]
63 void GCM_Shader::SetParameterM(CGparameter p, Matrix4& mat)
64 {
65    Matrix4 tempMatrix = transpose ( mat );
66    SetParameter (p, ( float *)& tempMatrix );
67 }
68
69 // Set a named paramter value
70 void GCM_Shader :: SetParameter (const char* name , float * data )
71 {
72    CGparameter p = GetParameter ( name );
73    if(p)
74    {
75      SetParameter(p, data );
76    }
77 }
78
79 char * GCM_Shader ::LoadBinaryFile(const char* name)
80 {
81    // Look for file
82    std::ifstream file (name, std::ios::in|std::ios::binary );
83    ASSERT_F((file != NULL),printf("Can't find file: %s\n",name));
84
85    // Load file attributes
86    file.seekg (0, std :: ios :: end );
87    unsigned int dataSize = (unsigned int)file.tellg ();
88    file.seekg (0, std :: ios :: beg );
89
90    // Reserve memory
91    char * data = ( char *) malloc ( dataSize );
92    // Copy file into memory
93    file.read (data , dataSize );
94    file.close (); // Done with the data , close the file.
95
96    return data;
97 }
```

2.2.3 GCMFragmentShader.h

Listing 7. GCM_FragmentShader.h

```
1 #pragma once
2 #include "GCM_Shader.h"
3
4 class GCM_FragmentShader: public GCM_Shader{
5
6    public:
7      //! Set a named paramter value
8      void SetParameter (CGparameter param, float * data );
9
10     //! Reloads the instruction cache of the fragment shader
11     void UpdateShaderVariables ();
12 };
```

2.2.4 GCMFragmentShader.cpp

Listing 8. GCM_FragmentShader.cpp

```
1 #include "GCM_FragmentShader.h"
2
3 // Reloads the instruction cache of the fragment shader
4 void GCM_FragmentShader :: UpdateShaderVariables ()
5 {
6    //Either this or cellGcmSetFragmentProgram() should be called
7    //when a parameter changes. However this command is more
8    //efficient as it only changes the parameters in memory.
9    cell::Gcm::cellGcmSetUpdateFragmentProgramParameter(offset);
10 }
11
12 // Set a named paramter value
13 void GCM_FragmentShader::SetParameter(CGparameter param,float↩
       * data)
14 {
15    cell::Gcm::cellGcmSetFragmentProgramParameter (program, ↩
       param, data, offset);
16 }
```

2.2.5 GCMVertexShader.h

Listing 9. GCM_VertexShader.h

```
1 #pragma once
2 #include "GCM_Shader.h"
3
4 class GCM_VertexShader: public GCM_Shader{
5
6    public:
7
8      //! Set a named paramter value
9      void SetParameter (CGparameter param, float * data );
10
11     //! Resolve position and colour parameter resource indexes.
12     void SetDefaultAttributes ();
13
14     //These indexes are used for cellGcmSetVertexDataArray()
15     // when sending data to the graphics chip. Essentially these
16     // represent where in the actual rendering hardware
17     // a particular parameter is bound.
18
19     //! Index of position parameter in vertex shader
20     uint32_t VERTEX_POSITION_INDEX;
21
22     //! Index of colour parameter in vertex shader
23     uint32_t VERTEX_COLOUR_INDEX;
24 };
```

2.2.6 GCMVertexShader.cpp

Listing 10. GCM_VertexShader.cpp

```
1 #include "GCM_VertexShader.h"
2
3 // Set a named paramter value
4 void GCM_VertexShader::SetParameter(CGparameter p, float* data)
5 {
6    cell::Gcm::cellGcmSetVertexProgramParameter (p, data );
7 }
8
9 // Resolve position and colour parameter resource indexes.
10 void GCM_VertexShader :: SetDefaultAttributes ()
11 {
12
13    CGparameter position_param =
14       cellGcmCgGetNamedParameter(program,"position");
15    //All vertex shaders should have a position parameter
16    //DBG_ASSERT(position_param);
17
18    CGparameter colour_param =
19       cellGcmCgGetNamedParameter(program,"color");
```

```
20
21   //Get the index of the vertex attribute that will be set for the vertex
22   // shader Turn parameters into resources,
23   // These are used for cellGcmSetVertexDataArray();
24   if(position_param  )
25   {
26     VERTEX_POSITION_INDEX=
27       cellGcmCgGetParameterResource(program,position_param)
28       -CG_ATTR0;
29   }
30
31   if(colour_param)
32   {
33     VERTEX_COLOUR_INDEX=
34       cellGcmCgGetParameterResource(program,colour_param)
35       -CG_ATTR0;
36   }
37 }
```

2.3 GCM Renderer

2.3.1 GCMRenderer.h

Listing 11. GCM_Renderer.h

```
1 #pragma once
2
3 #include <cell/gcm.h>   //for CellGcmSurface
4 #include "GCM_FragmentShader.h"
5 #include "GCM_VertexShader.h"
6
7 class GCM_Renderer{
8   private:
9     //! The current index of free space on local memory(Vram)
10    static unsigned int localHeapStart;
11
12    //! Set resolution and other things.
13    static void InitDisplay ();
14
15    //! Create Buffers
16    static void InitSurfaces ();
17
18    //! resolution.width*color_depth
19    static unsigned int color_pitch;
20
21    //! color_pitch * resolution.height
22    static unsigned int color_size;
23
24    //! resolution.width*z_depth;
25    static unsigned int depth_pitch;
26
27    //! depth_pitch * resolution.height
28    static unsigned int depthSize;
29
30    //! The current active render surface.
31    static unsigned char currentSurface;
32
33    //! The number of render surfaces/buffers.
34    //hardcode this data in for now.
35    static const unsigned char _numberOfSurfaces = 2;
36
37    //! The array of render surfaces.
38    static CellGcmSurface _surfaces[_numberOfSurfaces];
39
40    //! Blank constructor, static class
41    GCM_Renderer(){};
42
43  public:
44
45    //! The ratio width/height, of the ouput resolution
46    static float screenRatio;
47
48    //! Output screen Width, in pixels
49    static unsigned short screenWidth;
50
51    //! Output screen Height, in pixels
52    static unsigned short screenHeight;
53
54    //! Disguised constructor, calls the setup functions
55    static void start();
56
57    //! Shuts down Gcm.
58    static void shutdown();
59
60    // This function reserves a space of a specified size.
61    // Note: it doesn't actually write anything to memory.
62    // All it does is return the current address that points
63    // to free space, and then moves the localHeapStart
64    // by the size of the space needing reserved.
65
66    //! Reserves a space of a specified size on Local memory
67    static void* localMemoryAlloc (const unsigned int size );
68
69    //! Expands on Allocation() but also does some alignment
70    static void* localMemoryAlign (
71      const unsigned int alignment, const unsigned int size );
72
73    //! Set the active vertex and fragment shader
74    static void SetCurrentShader(
75        GCM_VertexShader& vert, GCM_FragmentShader& frag);
76
77    //! Sets cordinate/windows scaling and viewport stuff
78    static void SetViewport();
79
80    //! Sets appropriate render flags.
81    static void setupFrame();
82
83    //! Clears the Active buffer with a solid color.
84    static void clearSurface();
85
86    //! Show current buffer and set other buffer as the render target
87    static void swapBuffers ();
88 };
```

2.3.2 GCMRenderer.cpp

Listing 12. GCM_Renderer.cpp

```
1 #include "GCM_Renderer.h"
2 //#include <stdlib.h>
3 #include "asserts.h"
4 #include <cell/gcm.h>
5 #include <sysutil/sysutil_sysparam.h>
6 #include <iostream>  //memset
7
8 //The size of a chunk of main memory that the RSX can access.
9 //Has to be 1MB aligned, so minimum size is 1MB.
10 #define BUFFER_SIZE (1024*1024) //1MB
11
12 //Space reserved for each GCM command, minimum is 64KB.
13 #define COMMAND_SIZE (65536) // 64 KB
14
15 #define BUFFERS_COUNT (2) // double buffering
16
17 const unsigned char GCM_Renderer::_numberOfSurfaces;
18 unsigned short GCM_Renderer::screenHeight = 0;
19 unsigned short GCM_Renderer::screenWidth = 0;
20 unsigned char GCM_Renderer::currentSurface = 0;
21 unsigned int GCM_Renderer::localHeapStart = 0;
22 unsigned int GCM_Renderer::color_pitch = 0;
23 unsigned int GCM_Renderer::color_size  = 0;
24 unsigned int GCM_Renderer::depth_pitch = 0;
25 unsigned int GCM_Renderer::depthSize   = 0;
26 float GCM_Renderer::screenRatio  = 0.0f;
27 CellGcmSurface GCM_Renderer:: _surfaces[];
28
29 // Returns address of a continuous free memory segment of 'size'
30 void* GCM_Renderer::localMemoryAlloc(const unsigned int size)
31 {
32   unsigned int currentHeap = localHeapStart ;
33   localHeapStart += ( size + 1023) & (~1023);
34   return ( void *) currentHeap ;
35 }
36
37 // Expands on 'Allocation' function but also does some alignment
38 void* GCM_Renderer::localMemoryAlign(
```

```
39    const unsigned int alignment,const unsigned int size)
40 {
41    localHeapStart =
42      ( localHeapStart + alignment -1)
43      & (~( alignment -1));
44    return (void*)(localMemoryAlloc(size));
45 }
46
47 void GCM_Renderer::start()
48 {
49    localHeapStart = 0;
50    int err = 0;
51
52    void *host_addr = memalign(1024*1024, BUFFER_SIZE);
53    err = cellGcmInit(COMMAND_SIZE,BUFFER_SIZE,host_addr);
54    ASSERT_F((err == CELL_OK),
55      printf("#ERR cellGcmInit failed: 0x%x\n",err));
56
57    InitDisplay ();
58    InitSurfaces ();
59 }
60
61 void GCM_Renderer::shutdown()
62 {
63    int err = 0;
64
65    // Let RSX wait for final flip
66    cell::Gcm::cellGcmSetWaitFlip();
67
68    // Let PPU wait for all commands done (include waitFlip)
69    cell::Gcm::cellGcmFinish(0);
70    cell::Gcm::cellGcmFinish(1);
71
72    err = cellSysutilUnregisterCallback(0);
73    ASSERT_F((err == CELL_OK),
74    printf("#ERR cellSysutilUnregisterCallback failed 0x%x\n",err));
75 }
76
77 void GCM_Renderer::InitDisplay ()
78 {
79    CellVideoOutState videoState;
80    CellVideoOutResolution resolution;
81
82    //Get the current display mode,
83    // This has to have been previously set in the target manager
84    int err = cellVideoOutGetState(
85      CELL_VIDEO_OUT_PRIMARY, 0, &videoState);
86    ASSERT_F((err == CELL_OK),
87      printf("#ERR cellVideoOutGetState failed: 0x%x\n", err));
88
89    err = cellVideoOutGetResolution(
90      videoState.displayMode.resolutionId, &resolution);
91    ASSERT_F((err == CELL_OK),
92      printf("#ERR VideoOutGetResolution failed: 0x%x\n", err));
93
94    printf("Resolution:\t%ix%i\n",resolution.width,resolution.height)
95    screenWidth = resolution.width;
96    screenHeight = resolution.height;
97
98    //Rebuild a CellVideoOutConfiguration, using current resolution
99    uint32_t color_depth=4; // ARGB8
100   uint32_t z_depth=4;     // COMPONENT24
101   color_pitch = resolution.width*color_depth;
102   color_size =  color_pitch * resolution.height ;
103   depth_pitch = resolution.width*z_depth;
104   depthSize = depth_pitch * resolution.height ;
105
106   CellVideoOutConfiguration video_cfg ;
107   //Fill videocfg with 0
108   memset(&video_cfg , 0, sizeof(CellVideoOutConfiguration));
109
110   video_cfg.resolutionId = videoState.displayMode.resolutionId ;
111   video_cfg.format =
112   CELL_VIDEO_OUT_BUFFER_COLOR_FORMAT_X8R8G8B8;
113   video_cfg.pitch = color_pitch;
114
115   //Set the video configuration, we haven't changed anything
116   //other than possibly the Z/colour depth
117   err = cellVideoOutConfigure (
118     CELL_VIDEO_OUT_PRIMARY, &video_cfg , NULL , 0);
119   ASSERT_F((err == CELL_OK),
120     printf("#ERR cellVideoOutConfigure failed: 0x%x\n", err))
121
122   //Fetch videoState again, just to make sure everything went ok
123   err = cellVideoOutGetState(
124     CELL_VIDEO_OUT_PRIMARY, 0, &videoState);
125   ASSERT_F((err == CELL_OK),
126     printf("#ERR cellVideoOutGetState failed: 0x%x\n", err))
127
128   //Store the aspect ratio
129   switch (videoState.displayMode.aspect){
130    case CELL_VIDEO_OUT_ASPECT_4_3:
131      screenRatio = 4.0f/3.0f;
132      printf("Aspect ratio 4:3\n");
133      break;
134    case CELL_VIDEO_OUT_ASPECT_16_9:
135      screenRatio = 16.0f/9.0f;
136      printf("Aspect ratio 16:9\n");
137      break;
138    default:
139      printf("unknown ratio %x\n",videoState.displayMode.aspect);
140      screenRatio = 16.0f/9.0f;
141   }
142
143   cellGcmSetFlipMode ( CELL_GCM_DISPLAY_VSYNC );
144 }
145
146
147 void GCM_Renderer::InitSurfaces ()
148 {
149   printf("Creating buffers\n");
150
151   //GCMconfig holds info regarding memory and clock speeds
152   CellGcmConfig config ;
153   cellGcmGetConfiguration( &config );
154
155   //Get the base address of the mapped RSX local memory
156   localHeapStart = (uint32_t)config.localAddress;
157
158   //Allocate a 64byte aligned segment of RSX memory
159   void * depthBuffer = localMemoryAlign(64 , depthSize );
160   uint32_t depthOffset;
161
162   /*
163   cellGcmAddressToOffset converts an effective address in the area
164   accessible by the RSX to an offset value. An offset is the space
165   between from the base address of local memory and a certain
166   useable address. Offsets are used in gcm commands that deal with
167   shader parameters, texture mapping and vertex arrays. They serve
168   no real use other than as a parameter for these functions.
169   */
170
171   //The offset value will be stored into depthOffset.
172   cellGcmAddressToOffset ( depthBuffer , &depthOffset );
173
174
175   for( int i = 0; i < _numberOfSurfaces; ++i)
176   {
177     ///Allocate a 64byte aligned segment of RSX memory
178     // that is the size of a colour buffer
179     void *buffer = localMemoryAlign (64 , color_size );
180
181     //Get the offset address for it, store it in surfaces[i].colorOffset[0]
182     cellGcmAddressToOffset(buffer, &_surfaces[i].colorOffset[0]);
183
184     /*
185     This function registers a buffer that outputs to a display.
186     This is where the buffer is actually written to local memory.
187     Parameters:
188       cellGcmSetDisplayBuffer ( Buffer ID (0 - 7),
189       memory offset,
190       pitch - Horizontal byte width,
191       width - Horizontal resolution (number of pixels),
192       height - Vertical resolution(number of pixels)
193     */
194     cellGcmSetDisplayBuffer( i,_surfaces[i].colorOffset[0],
195         color_pitch,screenWidth,screenHeight);
196
```

```
    // Now we set other parameters on each CellGcmSurface object

    //where to place the color buffer, main memory or local memory
    _surfaces[i].colorLocation[0] =
          CELL_GCM_LOCATION_LOCAL;
    //Pitch size of the color buffer (resolution.width*color_depth)
    _surfaces[i].colorPitch[0] = color_pitch;
    //Target of the color buffer
    _surfaces[i].colorTarget = CELL_GCM_SURFACE_TARGET_0;

    //Init the color buffers
    //A CellGcmSurface can have 4 color buffers,but we only need 1
    for (int j = 1; j < 4; ++j)
    {
      _surfaces[i].colorLocation[j] =
        CELL_GCM_LOCATION_LOCAL;
      _surfaces[i].colorOffset[j]   = 0;
      _surfaces[i].colorPitch[j]    = 64;
    }

    //Type of render target (Pitch or swizzle)
    _surfaces[i].type = CELL_GCM_SURFACE_PITCH ;
    //Antialiasing format type (None in this case)
    _surfaces[i].antialias = CELL_GCM_SURFACE_CENTER_1;
    //Format of the color buffer
    _surfaces[i].colorFormat =CELL_GCM_SURFACE_A8R8G8B8;
    //Format of the depth and stencil buffers
    //Choice of 16-bit depth or 24-bit depth with an 8-bit stencil
    _surfaces[i].depthFormat=CELL_GCM_SURFACE_Z24S8;
    //where to place the depth buffer, main memory or local memory
    _surfaces[i].depthLocation=CELL_GCM_LOCATION_LOCAL;
    //Offset address of depth buffer (only need 1 for both surfaces)
    _surfaces[i].depthOffset = depthOffset;
    //Pitch size of the depth buffer (resolution.width*z_depth)
    _surfaces[i].depthPitch = depth_pitch;
    //Dimensions (in pixels)
    _surfaces[i].width = screenWidth ;
    _surfaces[i].height = screenHeight ;
    //Window offsets
    _surfaces[i].x = 0;
    _surfaces[i].y = 0;
  }

  /*
  The surfaces[] array contains CellGcmSurface objects and is in
  stack memory somewhere, and a bunch of new buffer objects have
  just been created and stored in RSX Local Memory.
  Each CellGcmSurface object has a pointer to its corresponding
  buffer in .colorOffset[0]. When we call cellGcmSetSurface(), we
  pass it an CellGcmSurface from our array, the parameters that we
  set on that object will be read, processed and passed to the RSX.
  */

  //Set Surface[0] to be the first surface to render to
  cell::Gcm::cellGcmSetSurface ( &_surfaces[0] );
  //Used to keep track of the surface currently being rendered to.
  currentSurface = 0;
}

// Set the active vertex and fragment shader
void GCM_Renderer::SetCurrentShader
  (GCM_VertexShader& vert, GCM_FragmentShader& frag)
{
 cell::Gcm::cellGcmSetFragmentProgram(frag.program,frag.offset);
 cell::Gcm::cellGcmSetVertexProgram(vert.program,vert.ucode);
}

// Initialises viewport (coordinate scaling)
void GCM_Renderer :: SetViewport () {
  uint16_t x,y,w,h;
  float min , max ;
  float scale [4] , offset [4];

  x = 0; // starting position of the viewport ( left of screen )
  y = 0; // starting position of the viewport (top of screen )
  w = screenWidth ; // Width of viewport
  h = screenHeight ; // Height of viewport
  min = 0.0f; // Minimum z value
  max = 1.0f; // Maximum z value

  // Scale our NDC coordinates to the size of the screen
  scale [0] = w * 0.5f;
  scale [1] = h * -0.5f; // Flip y axis !
  scale [2] = (max - min ) * 0.5f;
  scale [3] = 0.0f;

  // Translate from a range starting from -1 to a range starting at 0
  offset [0] = x + scale [0];
  offset [1] = y + h * 0.5f;
  offset [2] = (max + min ) * 0.5f;
  offset [3] = 0.0f;

  // Similar to the glViewport function, but with extra values
  cell::Gcm::cellGcmSetViewport(x,y,w,h,min,max,scale,offset);
}

void GCM_Renderer :: setupFrame ()
{
  cell::Gcm::cellGcmSetColorMask (
    CELL_GCM_COLOR_MASK_R |
    CELL_GCM_COLOR_MASK_G |
    CELL_GCM_COLOR_MASK_B |
    CELL_GCM_COLOR_MASK_A );

  cell::Gcm::cellGcmSetDepthTestEnable ( CELL_GCM_TRUE );
  //cellGcmSetDepthTestEnable ( CELL_GCM_FALSE );

  cell::Gcm::cellGcmSetDepthFunc ( CELL_GCM_LESS );
  //cellGcmSetDepthFunc(CELL_GCM_NEVER);

  cell::Gcm::cellGcmSetCullFaceEnable( CELL_GCM_FALSE );
  //cellGcmSetBlendEnable(CELL_GCM_FALSE);

  cell::Gcm::cellGcmSetShadeMode(CELL_GCM_SMOOTH);

  cell::Gcm::cellGcmSetBlendEnable(CELL_GCM_FALSE);

  // set polygon fill mode
  cell::Gcm::cellGcmSetFrontPolygonMode(
    CELL_GCM_POLYGON_MODE_FILL);

  //check for events
  //TODO: move this somewhere better
   cellSysutilCheckCallback();
}

void GCM_Renderer :: clearSurface ()
{
  cell::Gcm::cellGcmSetClearColor(
  (64 << 0)|(64 <<8)|(64 << 16)|(255 << 24));
  cell::Gcm::cellGcmSetClearSurface(
  CELL_GCM_CLEAR_Z | CELL_GCM_CLEAR_S |
  CELL_GCM_CLEAR_R |CELL_GCM_CLEAR_G |
  CELL_GCM_CLEAR_B | CELL_GCM_CLEAR_A );
}

// Switch which buffer is being rendered to and which is displayed
void GCM_Renderer :: swapBuffers ()
{
  //non-zero indicates hardware is still  processing the last flip
  while ( cellGcmGetFlipStatus ()!=0)
  {
    sys_timer_usleep (300);
  }

  //reset flips status
  cellGcmResetFlipStatus ();

  //flip
  cell::Gcm::cellGcmSetFlip (( uint8_t ) currentSurface);

  //flush the pipline
  cell::Gcm::cellGcmFlush ();
  cell::Gcm::cellGcmSetWaitFlip ();

  //Toggle the swapvalue flag
  if(currentSurface == 0){
    currentSurface = 1;
```

```
355   }else{
356     currentSurface = 0;
357   }
358
359   //Tell gcm to render into the correct surface
360   //TODO: make this suitable for more than 2 surfaces.
361   cell::Gcm::cellGcmSetSurface (&_surfaces[currentSurface]);
362 }
```

2.4 GCM Mesh Manager

2.4.1 GCMMeshManager.h

Listing 13. GCM_MeshManager.h

```
1 #pragma once
2 #include "Mesh.h"
3
4 //This class is for loading, decoding and rendering Mesh objects
5 class GCM_MeshManager{
6
7   public:
8     //! Render a Mesh with a supplied model view Projection matrix
9     static void render(stMesh* msh, Matrix4 mvp);
10    //! Load a mesh into the RSX local memory
11    static void loadOnGPU(stMesh* msh);
12 };
```

2.4.2 GCMMeshManager.cpp

Listing 14. GCM_MeshManager.cpp

```
1 #include "GCM_MeshManager.h"
2 #include "GCM_Renderer.h"
3 #include "asserts.h"
4
5 void GCM_MeshManager::loadOnGPU(stMesh* msh)
6 {
7   printf("Loading mesh on RSX, verts:%i\n", msh->numVerts);
8   //reserve local memory
9   msh->vertexBuffer =
10  (stVertex*) GCM_Renderer::localMemoryAlign
11  (128, sizeof(stVertex) * msh->numVerts);
12
13  //load data into VB
14  for (int i=0; i< (msh->numVerts); ++i)
15  {
16    msh->vertexBuffer[i].x = msh->vertexData[i].x;
17    msh->vertexBuffer[i].y = msh->vertexData[i].y;
18    msh->vertexBuffer[i].z = msh->vertexData[i].z;
19    msh->vertexBuffer[i].rgba = msh->vertexData[i].rgba;
20  }
21
22  //calculate offset
23  int err = cellGcmAddressToOffset(
24  (void*)msh->vertexBuffer, &msh->vertexBufferOffset);
25  ASSERT_M((err==CELL_OK),"GcmAddressToOffset failed");
26
27  msh->loadedLocal = true;
28 }
29
30
31 void GCM_MeshManager::render(stMesh* msh, Matrix4 mvp)
32 {
33  ASSERT(msh->loadedLocal);
34  //set active shader
35  GCM_Renderer::SetCurrentShader
36    (*msh->vertShader, *msh->fragShader);
37
38  //give vertex data to the shader
39  cell::Gcm::cellGcmSetVertexDataArray(
40      msh->vertShader->VERTEX_POSITION_INDEX,   //index
41      0, //Frequency
42      sizeof(stVertex), //stride
43      3, //size
44      CELL_GCM_VERTEX_F, //type
45      CELL_GCM_LOCATION_LOCAL, //location
46      msh->vertexBufferOffset //offset
47    );
48
49  cell::Gcm::cellGcmSetVertexDataArray(
50      msh->vertShader->VERTEX_COLOUR_INDEX,
51      0,
52      sizeof(stVertex),
53      4,
54      CELL_GCM_VERTEX_UB,
55      CELL_GCM_LOCATION_LOCAL,
56      msh->vertexBufferOffset + sizeof(float)*3
57    );
58
59  msh->vertShader->SetParameterM("modelViewProj",mvp);
60
61  //not sure wether to call this?
62  //FS->UpdateShaderVariables();
63
64  //draw arrays
65  if (msh->strip){
66    cell::Gcm::cellGcmSetDrawArrays(
67      CELL_GCM_PRIMITIVE_TRIANGLE_STRIP,
68      0, msh->numVerts);
69  }else{
70    cell::Gcm::cellGcmSetDrawArrays(
71      CELL_GCM_PRIMITIVE_TRIANGLES,
72      0, msh->numVerts);
73  }
74 }
```

2.5 Torus Generator

This section of code generates an array of vertices that form a torus(doughnut). The points are in a striped format, and the colours are generated randomly. This code is not specific to the PS3, however the colours are in a ABGR format, as this is how CG shaders read colours. If you are porting this to opengl you would have to swap the endian mode to be RGBA.

2.5.1 TorusGenerator.h

Listing 15. TorusGenerator.h

```
1 #pragma once
2 #include <vector>
3 #include <cmath>
4 #include "Common.h"
5 #include "asserts.h"
6
7 inline int randomColor()
8 {
9   int x = rand() & 0xff;
10  x |= (rand() & 0xff) << 8;
11  x |= (rand() & 0xff) << 16;
12  x |= (rand() & 0xff) << 24;
13
14  return x;
15 }
16
17 inline std::vector<stVertex> CreateTorus(float InnerRadius,
18 float OuterRadius, unsigned int Sides, unsigned int Rings)
19 {
20  ASSERT_M((Sides >= 3), "Sides must be 3 or bigger");
21  ASSERT_M((Rings >= 3), "Rings must be 3 or bigger");
22  ASSERT_M((InnerRadius >=0),"InnerRadius can't be negative");
23  ASSERT_M((OuterRadius>InnerRadius),"OuterRadius !>Inner")
24
25  const float centerRadius = InnerRadius *0.5f +OuterRadius *0.5f;
26  const float rangeRadius = OuterRadius - centerRadius;
27
28  std::vector<stVertex> torus;
29  const float stepRing = (360.0f / Rings) * (float)(M_PI/180.0f);
30  const float stepSide = (360.0f / Sides) * (float)(M_PI/180.0f);
31  for(unsigned int i = 0; i < Rings; i++)
32  {
```

```
    const float curRings[2] =
    {
      stepRing * (i + 0),
      stepRing * (i + 1)
    };
    const float ringSins[2] ={sinf(curRings[0]),sinf(curRings[1])};
    const float ringCoss[2] ={cosf(curRings[0]),cosf(curRings[1])};

    for(unsigned int j = 0; j < Sides+1; j++)
    {
      const float curSide = (j % Sides) * stepSide;
      const float sideSin = sinf(curSide);
      const float sideCos = cosf(curSide);

      for(int k = 0; k < 2; k++)
      {
        stVertex vert;

        vert.x=(centerRadius+rangeRadius*(sideCos))*ringCoss[k];
        vert.y = (rangeRadius * sideSin);
        vert.z = (centerRadius+rangeRadius*(sideCos))*ringSins[k];
        vert.rgba = randomColor();
        // If we want textures?
        // instead of 0-->1 we go from 1-->0
        // u = 1.0f - ((1.0f / Rings) * (i+k));
        // v = ((1.0f / Sides) * (j));

        torus.push_back(vert);
      }
    }
  }

  return torus;
};
```

2.6 Main

The main file doesn't contain any complex new code. We are generating matrices just before and during the render loop, all the hard work is being done by the maths library so it's just a case of plugging in numbers. An interesting addition is the sysutil_callback function. This allows us to listen for system events, such as the home menu being opened, or a system window being rendered. We are only acting on one event, CELL_SYSUTIL_REQUEST_EXITGAME. This is called when a user opens the home menu via the home button on the controller and choses to close the game.

2.6.1 main.cpp

Listing 16. main.cpp

```
#include "asserts.h"
#include "GCM_Renderer.h"
#include "Mesh.h"
#include "TorusGenerator.h"
#include "GCM_MeshManager.h"

#include <stdio.h>   //Printf
#include <sysutil/sysutil_sysparam.h> //sysutil_callback
#include <vectormath/cpp/vectormath_aos.h>   //Matrix4
using namespace Vectormath::Aos;

//-- Function definitions
  //! System event callback manager
  static void sysutil_callback (unsigned long long status,
    unsigned long long param, void *userdata);

//-- Globals
  //! Flag to signal the gameloop should stop
  static bool run = false;

// ************************************************
// Program Entry Point: main
int main()
{
  printf("\n\n------- Program Started -------\n\n");

  int err = cellSysutilRegisterCallback( 0, sysutil_callback, NULL );
  ASSERT_F((err == CELL_OK),
    printf("#ERR cellSysutilRegisterCallback failed: 0x%x\n", err));

  GCM_Renderer::start();

  // Load shaders ------------------#
  printf("Making Shaders\n");
  GCM_VertexShader VS;
  GCM_FragmentShader FS;

  FS.LoadBinaryShader("/app_home/fs_basic.fpo");
  FS.initProgram(true);
  VS.LoadBinaryShader("/app_home/vs_basic.vpo");
  VS.initProgram(false);

  VS.SetDefaultAttributes();

  printf("Shaders parsed\n");

  // Set current shader
  GCM_Renderer::SetCurrentShader(VS,FS);

  // Create mesh data to render ----------------------#

  stMesh torus;
  torus.vertexData = CreateTorus( 1, 5, 24, 16 );
  torus.loadedMain = true;
  torus.numVerts = torus.vertexData.size();
  torus.strip = true;
  torus.fragShader = &FS;
  torus.vertShader = &VS;

  GCM_MeshManager::loadOnGPU(&torus);

  //Setup view matricies ------------------------#
  Point3 CameraPos = Point3(0,-20,0);
  Point3 CameraLook = Point3(0,0.1f,0.1f);
  Vector3 UpVector =  Vector3(0,1,0);

  //Projection matrix: 60deg Field of View, display range: 0.1 to 1000
  Matrix4 projMatrix = Matrix4::perspective
    (60.0f*(M_PI/180), GCM_Renderer::screenRatio , 0.1f, 1000.0f);
  Matrix4 viewMatrix = Matrix4::lookAt
    (CameraPos , CameraLook, UpVector);
  Matrix4 ViewProjection = (projMatrix * viewMatrix);

  //Torus ModelProjection
  Matrix4 bigtorusModelProjection =
    Matrix4::scale(Vector3(2.0f, 2.0f, 2.0f));

  Matrix4 sidetorusModelProjection =
    Matrix4::translation(Vector3(15.0f,5.0f,7.0f)) *
    Matrix4::scale(Vector3(1.25f, 1.25f, 1.25f));

  //-------------------------------------------
  //---------- Start Rendering ----------#
  float a;
  run = true;
  while(run)
  {
    sys_timer_usleep(500);
    //Prep for render
    GCM_Renderer::SetViewport();
    GCM_Renderer::setupFrame();
    GCM_Renderer::clearSurface();

    //--- Render
      a+= 1.01f;
      Matrix4 spinX = Matrix4::rotation
          (a*0.75f*(M_PI/180), Vector3(1,0,0));
      Matrix4 spinY = Matrix4::rotation
          (a*(M_PI/180), Vector3(0,1,0));
      Matrix4 spinZ = Matrix4::rotation
          (a*0.25f*(M_PI/180), Vector3(0,0,1));
```

```
102
103         Matrix4 mvp =
104           ViewProjection * bigtorusModelProjection * spinZ* spinY;
105         GCM_MeshManager::render(&torus,mvp);
106
107         mvp =
108           ViewProjection * sidetorusModelProjection * spinX * spinY;
109         GCM_MeshManager::render(&torus,mvp);
110       //---
111
112       //Swap buffers
113       GCM_Renderer::swapBuffers();
114     }
115
116     printf("\n -- Shutting Down --\n");
117     GCM_Renderer::shutdown();
118
119     printf("\n  -------- Quitting --------\n");
120 }
121
122 void sysutil_callback(uint64_t status, uint64_t param, void *userdata)
123 {
124     printf("System Event! %#08x\n",status);
125     if (status == CELL_SYSUTIL_REQUEST_EXITGAME) {
126         printf("System has requested EXITGAME\n");
127         run = false;
128     }
129 }
```

3. Conclusion

In summary, if everything went well, you should have got graphics working on your PS3 and are ready to start rendering complex geometry (e.g., virtual environments). The PS3 SDK and Visual Studio integration should be work seamlessly - so that you can step through and debug your compiled PS3 code in the SN debugger.

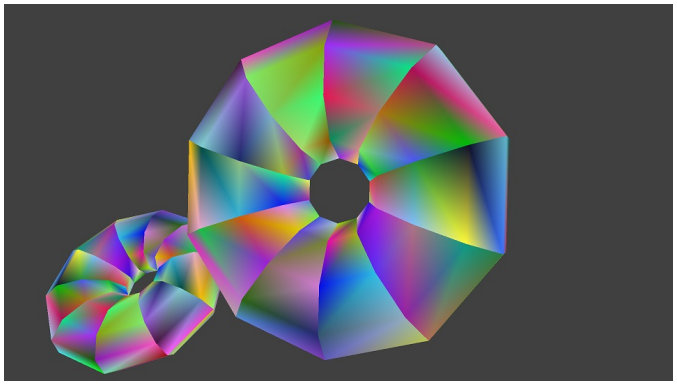

Figure 1. Rendered output - Screen capture showing two procedurally generated torus shapes with interpolated random vertex colours.

Recommended Reading

Programming the Cell Processor: For Games, Graphics, and Computation, Matthew Scarpino, ISBN: 978-0136008866

Vector Games Math Processors (Wordware Game Math Library), James Leiterman, ISBN: 978-1556229213

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

References

[1] Edinburgh Napier Game Technology Website. www.napier.ac.uk/games/. *Accessed: Feb 2014*, 2014. 1