Tutorial 4
Introduction to GCM and PS3 graphics

Playstation 3 Development

Sam Serrels and Benjamin Kenwright[*]

Abstract
A beginners guide to getting started with graphical programming and developing on Sony's Playstation 3 (PS3). This article gives a brief introduction for students to initializing and working with the GCM graphics API. For example, setting up the GCM, getting and setting screen paramaters, initializing basic vertex/pixel shaders, and drawing triangles.

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. Students have constant access to he Sony DevKits and encourage enthusiastic students to design and build their own games and applications during their spare time [4].

This tutorial This tutorial will cover the essentials to working with the Sony GCM rendering library and interfacing with the RSX graphics processor and it's memory. Displaying any sort of graphics on a screen from the Playstation 3 requires knowledge of how to to work it's specific graphics hardware. The limited memory of the system brings in an extra dimension of work when managing and transferring data, and calls for a greater level of optimisation than desktop computer graphics. This tutorial will cover how to work with bare-minimum memory management provided to make a usable and understandable rendering framework.

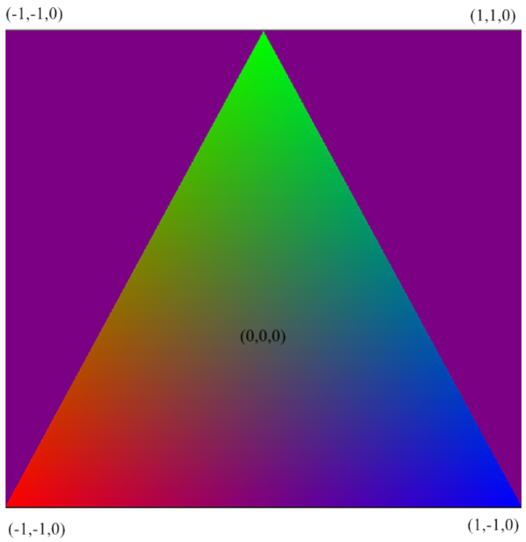


Figure 1. Screen Capture - This tutorial draws a triangle to the screen. We set the view and projection matrix to an identity to keep the example and simple as possible.

Additional Reading In addition to the lesson tutorials, we would recommend reading a number of books on Playstation 3 development and cross-platform coding, such as, Cell Programming for the PS3 [3], Vector Maths and Optimisation for the PS3 [1], and Cross-Platform Development in C++ [2].

1. Graphics Command Management (GCM)

What is GCM and why do use GCM for the graphics? GCM is the Graphics Command Management library (i.e., libGCM). We use GCM as it has no abstraction layer and allows us to generates graphical commands directly (i.e., for computational speed reasons). This article shows you the essential API necessary to manage the graphical command generation and command buffers to control the command chain and display graphics on the screen.

1.1 GCM and PSGL

Developing with the official SDK leaves you with two APIs to choose from in terms of rendering. GCM and PSGL (i.e., Playstation OpengGL). GCM is specific to the hardware and is as low level as it gets, and as a result what you make with it will (or should) preform somewhat better. However, it should be noted, the PSGL is also popular due to using the OpenGL convention - hence simple to understand and implement.

OpenGL ES 1.0 PSGL is OpenGL ES 1.0 complaint meaning that if you're a beginner there are tons of resources available online with information about writing for it. Furthermore, it means that you wont be teathering yourself to PS3 API. OpenGL exists in some form or another on effectively every platform in existence so its a good idea to become familiar with it and it will make it a lot easier to port anything you write. PSGL also supports a lot of stuff that isn't a standard part of OpenGL ES 1.0 like vertex buffer objects and NVIDIA Cg shaders. While this article introduces GCM - as it offers the most flexibility and power - we will introduce the PSGL later. When working with the PSGL, there is no need to start completely from scratch, as you do with the GCM. However, as with every coin - there are two sides - with the added simplicity you loose the additional control and speed.

2. GCM Memory Management

GCM does no memory management for us. During initialization of the GCM library we're given a pointer to the start of the RSX memory. That means we're going to have to create our own functions to manage the graphical RAM (e.g., allocate and de-allocation of textures and shaders). GCM calls graphics RAM 'local memory' - as it's local to the RSX processor of the Playstation 3. In addition, specific GCM functions require structures to memory aligned (e.g., 32, 64 or 128 byte alignment), so we must provide functions to allocate 'aligned' memory.

2.1 Memory Allocation

So we have have established that GCM does no memory management of the local memory, but what does that mean? How much work are we going to have to do really? Let us have a look at how main memory is managed (See Figure 2) :

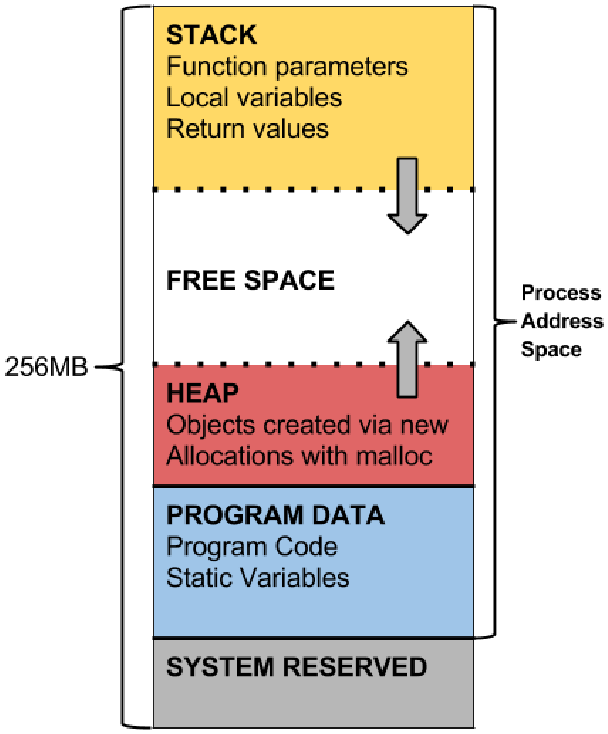


Figure 2. Main Memory - How memory is allocated

Stack memory The stack has the best kind of management, it's completely handled by the system. When you create a vriable such as "int var = 0;" you don't need to care where it is stored (although with pointers you can find out) and you don't have to free the memory once you have finished with the variable, this is done automatically when it goes out of scope.

Heap memory The Heap is different to the stack as you need to keep track of what is allocated, and delete/free that memory when it is no longer needed. You don't need to know *where* in the heap items are placed when they are created, the system will find an appropriate section of free space, reserve it, and then give you a pointer to the start of this space so you can reference and delete the object.

RSX local memory Local memory is just like heap memory, but it doesn't keep track of anything. All it knows is the range of memory addresses that are valid. It doesn't know if anything is stored in any of these addresses, just that items *can* be stored there. We must keep track of everything we store in local memory, and how big each item is, so when we don't overwrite previous data when storing new data. If we wanted to we could write anything to any valid local memory address and overwrite any previous data, this isn't allowed in main memory.

2.2 Memory Access

If we need to manage local memory, how do we access it?
The PPU can write and read directly to RSX local memory, this is done by requesting a pointer to the start of the local memory. GCM grantees that this memory is contiguous, so the next address up from the start address is valid, and the next one and so on until the maximum memory size is reached.
The addresses that we use to access the local memory from the main application code running on the PPU, are the result of some mapping functions. To the PPU, the starting address of local memory is around the what would be the address for 1GB, this is to avoid clashes with main memory, as main memory address will never be bigger than 256mb.
The RSX can't actually use these mapped addresses to access it's own memory as the true starting address of its memory is 0, not 1GB. So when code is running explicitly on the RSX chip (e.g. Shaders) it needs to use a different addressing system than the PPU.
The RSX uses offset addresses, which is how far from the start of local memory an item is located. This may sound confusing, but keep in mind that offset address are only needed in RSX specific code, and there is a function that converts between the addressing systems. (See Figure 3)

Reading Main memory from RSX This system is not symmetrical, the RSX cannot access main memory in the same mannor, it doesn't have all of the main memory addresses mapped to its own address system. It does however have *some* of the main memory address mapped, this is a small section of memory that is used to send instructions to the RSX.
During initialisation of GCM, the size and location of this RSX accessible "chunk" of main memory must be defined. This chunk is used as as the Command buffer, when you call a GCM command, it is stored in this buffer to be read and executed by the RSX. If the RSX needs to access other data from main memory, it must call specific data transfer functions, which have additional overhead, however this is a rare occasion as the PPU almost always in charge of sending the data to the RSX. The only time this may happen is for streaming images or large arrays.

2.3 GCM Memory code

This code reserves a 1MB area of main memory for storing GCM commands, each command is 64kb in size. This means that the buffer can store a queue of 16 commands, but in reality 15, as GCM uses the first 4kb of the buffer for special flags.

Listing 1. Initialising GCM with a command buffer

```
//The size of a chunk of main memory that the RSX can access.
//Has to be 1MB aligned, so minimum size is 1MB.
# define HOST_SIZE (1024*1024) //1MB

// Space reserved for each GCM command, minimum is 64KB.
# define COMMAND_SIZE (65536) // 64 KB

//Reserve a 1MB aligned chunk of memory. This is happening
// on main memory, so we can use the stock memalign() function.
void *host_addr = memalign(1024*1024, HOST_SIZE);

//Initialize libgcm and map the command buffer
// from main memory to the RSX IO address space.
cellGcmInit ( COMMAND_SIZE , HOST_SIZE , host_addr);
```

Here are the functions that we will use to allocate data into local memory:

Listing 2. Managing our own memory on the RSX

```
//The Pointer to the Start of our chunk of RSX accessible memory
uint32_t localHeapStart = 0;

//At some point, once GCM has been initialised, we will do this:
localHeapStart = (uint32_t)config.localAddress;
//Now that we know the stating address of local memory,
// we can start writing to it. As we write data, we
// increment localHeapStart, like a bookmark,
// so we don't overwrite data.

//This function reserves a space of a specified size.
//Note: it doesn't actually write anything to memory.
//All it does is return the current address that points to free space,
//then moves localHeapStart to after the space needing reserved.
void * LocalMemoryAlloc ( const uint32_t size )
{
  uint32_t currentHeap = localHeapStart ;
  localHeapStart += ( size + 1023) & (~1023);
  return ( void *) currentHeap ;
}

// Expands 'Allocation' function but includes byte alignment
void* LocalMemoryAlign(const uint32_t alignment,
      const uint32_t size)
{
  localHeapStart =
    (localHeapStart + alignment −1) & (~( alignment −1));
  return ( void *) LocalMemoryAlloc ( size );
}
```

3. GCM Libraries

Linker Input Dependencies We use external libraries (e.g., libGCM), hence, we need to let the compiler know where they are. See Figure 4 to ensure the additional dependencies tab within the Visual Studio configuration is setup to include the necessary paths and libraries .

Listing 3. Visual Studio PS3 include libraries - see Figure 4

```
libgcm_cmddbg.a;
libgcm_sys_stub.a;
libsysutil_stub.a;
```

These are also necessary, but should already be included by default

```
libsn.a
libm.a
libio_stub.a
libfs_stub.a
```

The header files we will be using in this tutorial are: sysutil/sysutil_sysparam.h, cell/gcm.h, vectormath/cpp/vectormath_aos.h, stdio.h, stdlib.h, math.h, string,

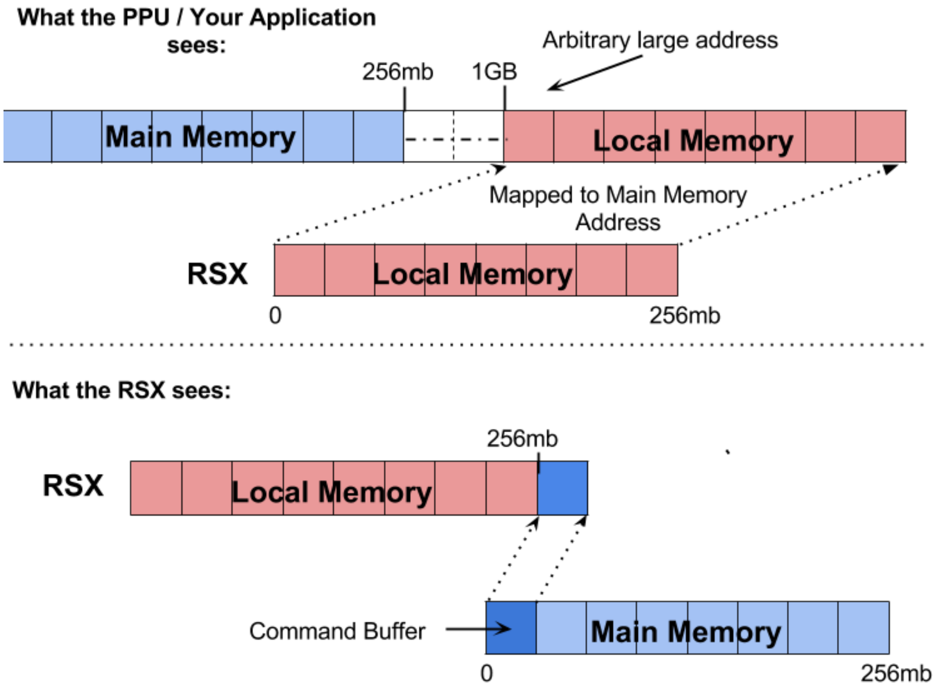
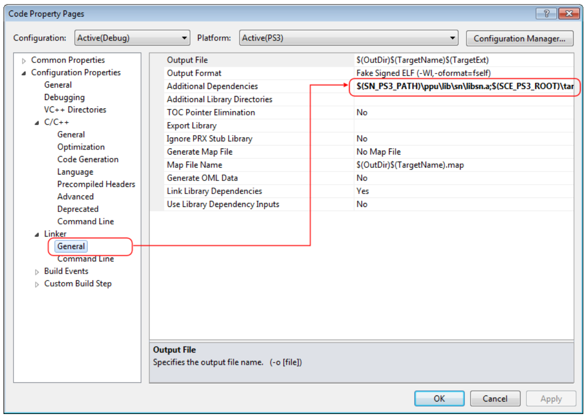


Figure 3. **GCM Memory Mapping** - How RSX local memory is mapped to Main memory



Configuration: Active(Debug)   Platform: Active(PS3)

Output File	$(OutDir)$(TargetName)$(TargetExt)
Output Format	Fake Signed ELF (-Wl,-oformat=fself)
Additional Dependencies	**$(SN_PS3_PATH)\ppu\lib\sn\libsn.a;$(SCE_PS3_ROOT)\tar**
Additional Library Directories	
TOC Pointer Elimination	No
Export Library	
Ignore PRX Stub Library	No
Generate Map File	No Map File
Map File Name	$(OutDir)$(TargetName).map
Generate OML Data	No
Link Library Dependencies	Yes
Use Library Dependency Inputs	No

Output File
Specifies the output file name. (-o [file])

Figure 4. **Configure Project Link Libraries** - Ensure you have the necessary libraries to compile GCM.
In the case that Visual studio cannot find the libraries, add these paths to Additional Library Directories :
$(SCE_PS3_PATH)/ppu/lib
$(SCE_PS3_ROOT)/target/ppu/lib

4. Compiling Cg Shaders (.cg, .vpo, .fpo)

We need to can compile any '.cg' files to their compiled raw binary format for the Playstation 3 graphical output. The PS3 has no 'default' shaders built in. Hence, we need to include a minimum binary shader to get something displaying on screen.

Automation In later lessons, we will automatically include the compilation of the shaders using Visual Studio's 'Custom Build Step'. For now, we show how to compile a shader at the command prompt to create a simple shader binary that can be loaded either using the standard C libraries (e.g., fopen and fread), or included at the bottom of the file (as done in this example) - so we don't need to include any external assets.

Command Prompt The PS3 SDK installation is installed at "C:/usr/local/cell/" - and is defined by "$(SCE_PS3_ROOT)". Within the PS3 SDK there is an executable called 'sce-cgc.exe' that we use to compile the shaders into their specific binary format. We can call the shader compiler from the command prompt and pass it the necessary arguments to create the compiled vertex and fragment shader, as shown below in Listing 4 with the basic shader text files given in Listing 5 and 6.

Listing 4. Compiling the fragment and vertex shader (fs_basic.cg) at the command prompt (note the profile information 'sce_vp_rsx' and 'sce_fp_rsx')

```
C:\usr\local\cell\host−win32\Cg\bin\sce−cgc.exe −profile sce_fp_rsx −o ".\↩
        fs_basic.fpo" ".\fs_basic.cg"

// Command prompt output:
// 11 lines, 0 errors.
// Binary shader output to C:\napier\sourcecode\fs_basic.fpo

C:\usr\local\cell\host−win32\Cg\bin\sce−cgc.exe −profile sce_vp_rsx −o "↩
        .\vs_basic.fpo" ".\vs_basic.cg"


// Command prompt output:
// 17 lines, 0 errors.
// Binary shader output to C:\napier\sourcecode\vs_basic.fpo
```

Listing 5. Basic Vertex Shader (vs_basic.cg)

```
// vs_basic.cg
void main
(
  float4 position : POSITION,
  float4 color  : COLOR,

  uniform float4x4 modelViewProj,

  out float4 oPosition : POSITION,
  out float4 oColor    : COLOR
)
{
  oPosition = position; // mul(ModelViewProjMatrix, position);
  oColor = color;
}
```

Listing 6. Basic Fragment Shader (fs_basic.cg)

```
// fs_basic.cg
void main
(
  float4 color_in      : COLOR,
  out float4 color_out : COLOR
)
{
  color_out = color_in;
}
```

5. Skeleton Graphics 'Without' Any Wrapper Classes

We present a single file that shows the sequential steps necessary from entering 'main()' all the way through to getting triangles on screen (e.g., initializing monitor, allocating system resources, loading in graphical components, such as the vertex and pixel shader).

5.1 Implementation Overview

The sample GCM implementation shown below in Listing 7 will get you started rendering using the Playstation GCM API. The implementation is stripped down to the low-level API (i.e., stripped out any wrapper classes). Furthermore, the sample listing is a single function (i.e., main()) that the student can work through from start to finish to see the steps necessary. This tutorial focuses on introducing the API necessary to render graphics on the Playstation. Later, tutorials will work on loading complex geometry and animations. Note, the Playstation SDK comes with the vector, quaternion, and matrix classes for mathematical operations (e.g., creating camera matrices, matrix multiplication, dot product).

The basic GCM program shown below in Listing 7 performs the following steps:

- Initialise GCM and display
- Load in our pre-compiled shaders
- Initialise a vertex array (i.e., triangle information)
- Render the scene while updating the triangles (i.e., show them changing)
- Release resources and exit

The implementation can be a bit over-whelming initially - since it requires over 400 lines of code to get a simple triangle on the screen. However, once we explain everything in detail and you understand what is happening at each stage, you'll realize the flexibility and potential advantages of being able to create and control low-level features.

5.2 Source Code

Listing 7. Complete implementation - main.cpp - self contained graphical source code example to get you up and running with the PS3 renderer quickly - uses GCM library

```
/*
 Self-contained stripped down - skeleton system to get you up and running with graphics on the PS3 - no wrapper classes,
 no classes or compiling shaders - single file - that does everything from start to end - initializes gcm,
 sets up memory, clears screen to gradually changing colour, and draws a triangle on the screen
*/
// Custom Asserts - For Debugging
#define DBG_HALT { __asm__ volatile( "trap" ); }

#define DBG_ASSERT(exp) { if ( !(exp) ) {DBG_HALT;} }
// Prints the suplied string on assert fail, then call DBG_HALT
#define DBG_ASSERT_MSG( exp, smsg ) { if ( !(exp) ) {puts (smsg); DBG_HALT;} }
// Calls the suplied function on assert fail, then call DBG_HALT
#define DBG_ASSERT_FUNC( exp, func) { if ( !(exp) ) {func; DBG_HALT;} }

// For putf - prints
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <string> // for memcpy(..)
using namespace std;

// A very basic vertex and framgent shader - placed at the bottom of the file
extern unsigned char fs_basic[208];
extern unsigned char vs_basic[672];

// ***********************************

// e.g., CELL_VIDEO_OUT_PRIMARY, CellVideoOutResolution
#include <sysutil/sysutil_sysparam.h>

// libgcm
#include <cell/gcm.h>
using namespace cell::Gcm;

// vectormath - so we can use Matrix4
#include <vectormath/cpp/vectormath_aos.h>
using namespace Vectormath::Aos;


/*
  To enable the RSX to access main memory, we must setup a section
  of 1MB aligned memory within main memory and point RSX at it.
  When you call a GCM command, it is stored in a part of this memory chunk
  called the command buffer, which the RSX reads from and executes.
*/

//The size of a chunk of main memory that the RSX can access.
//Has to be 1MB aligned, so minimum size is 1MB.
# define HOST_SIZE (1024*1024) //1MB
//size of the space reserved for each GCM command, minimum size is 64KB.
# define COMMAND_SIZE (65536) // 64 KB

# define BUFFERS_COUNT (2) // double buffering

// We need to manage our own memory on the PS3 - furthermore,
// we have to ensure the memory we create is aligned on specific
// boundaries, e.g., 64, 128.  We, set the start address of our local
// heap on startup

// API call necessary to get the start address after we've set
// everything up:
// CellGcmConfig configa ;
// cellGcmGetConfiguration( &configa );
// localHeapStart = (uint32_t)configa.localAddress;

// Start of our chunk of RSX accessible memory
uint32_t localHeapStart = 0;

//This function reserves a space of a specified size. Note: it doesn't actually write anything to memory.
//All it does is return the current address that points to free space,
// and then moves the localHeapStart by the size of the space needing reserved.

// Allocation, returns address to the start of a continuous memory segment of 'size'
void * LocalMemoryAlloc ( const uint32_t size )
{
  uint32_t currentHeap = localHeapStart ;
  localHeapStart += ( size + 1023) & (~1023);
  return (void *) currentHeap ;
}

// Expands on 'Allocation' function but also does some alignment
void * LocalMemoryAlign (const uint32_t alignment, const uint32_t size )
{
  localHeapStart = ( localHeapStart + alignment -1) & (~( alignment -1));
  return ( void *) LocalMemoryAlloc ( size );
}

// Vertex structure - very simple
struct stVertex
{
  float x, y, z;
  uint32_t rgba ;
};


// Heart of 'everything' for this introduction -
// get feel for the essential API - step-by-step
//
// Program Entry Point: main
//
int main()
{
  puts("Program Entry Point: main\n");

  //Reserve a 1MB aligned chunk of memory
  void *host_addr = memalign(1024*1024, HOST_SIZE);
  DBG_ASSERT_MSG(host_addr != NULL,"memalign() failed!");

  //This function initializes libgcm and maps the buffer on main memory to IO address space so that RSX can access it.
  int err = cellGcmInit ( COMMAND_SIZE , HOST_SIZE , host_addr);
  DBG_ASSERT_MSG( err==CELL_OK, "cellGcmInit failed!" );

  // *** #Init Display# ************************

  CellVideoOutState videoState;
  CellVideoOutResolution resolution;

  //Get the current display mode,
  // This has to have been previously set in the target manager at some point
  err = cellVideoOutGetState(CELL_VIDEO_OUT_PRIMARY, 0, &videoState);
  DBG_ASSERT_MSG( err==CELL_OK, "cellVideoOutGetState failed !" );

  err = cellVideoOutGetResolution(videoState.displayMode.resolutionId, &resolution);
  DBG_ASSERT_MSG( err==CELL_OK, "cellVideoOutGetResolution failed !" );

  printf("Output Resolution:\t%i x %i \n",  resolution.width, resolution.height);

  //Rebuild a CellVideoOutConfiguration, using the current resolution
  uint32_t color_depth=4; // ARGB8
  uint32_t z_depth=4;     // COMPONENT24
  uint32_t color_pitch = resolution.width*color_depth;
  uint32_t color_size =  color_pitch * resolution.height ;
  uint32_t depth_pitch = resolution.width*z_depth;
  uint32_t depthSize = depth_pitch * resolution.height ;

  CellVideoOutConfiguration video_cfg ;
  //Fill videocfg with 0
  memset(&video_cfg , 0, sizeof(CellVideoOutConfiguration));

  video_cfg.resolutionId = videoState.displayMode.resolutionId ;
  video_cfg.format = CELL_VIDEO_OUT_BUFFER_COLOR_FORMAT_X8R8G8B8 ;
  video_cfg.pitch = color_pitch;

  //Set the video configuration, we haven't changed anything other than possibly the Z/colour depth
  err = cellVideoOutConfigure ( CELL_VIDEO_OUT_PRIMARY, &video_cfg , NULL , 0);
  DBG_ASSERT_MSG( err==CELL_OK, "cellVideoOutConfigure failed !" );

  //Fetch videoState again, just to make sure everything went ok
  err = cellVideoOutGetState(CELL_VIDEO_OUT_PRIMARY, 0, &videoState);
  DBG_ASSERT_MSG( err==CELL_OK, "cellVideoOutGetState failed !" );

  //Store the aspect ratio
  float screenRatio;
  switch (videoState.displayMode.aspect){
    case CELL_VIDEO_OUT_ASPECT_4_3:
      screenRatio = 4.0f/3.0f;
      break;
    case CELL_VIDEO_OUT_ASPECT_16_9:
      screenRatio = 16.0f/9.0f;
      break;
    default:
      printf("unknown aspect ratio %x\n", videoState.displayMode.aspect);
      screenRatio = 16.0f/9.0f;
  }

  cellGcmSetFlipMode ( CELL_GCM_DISPLAY_VSYNC );


  // *** #Create buffers# **************************
  printf("Creating buffers\n");

  //GCMconfig holds info regarding memory and clock speeds
  CellGcmConfig config ;
  cellGcmGetConfiguration( &config );

  //Get the base address of the mapped RSX local memory
  localHeapStart = (uint32_t)config.localAddress;

  //Allocate a 64byte aligned segment of RSX memory that is the size of a depth buffer
  void * depthBuffer = LocalMemoryAlign(64 , depthSize );
  uint32_t depthOffset;

  /* cellGcmAddressToOffset converts an effective address in the area accessible by the RSX to an offset value.
    An offset is the space between from the base address of local memory and a certain useable address.
    Offsets are used in gcm commands that deal with shader parameters, texture mapping and vertex arrays.
    They serve no real use other than as a parameter for these functions.
  */

  //The offset value will be stored into depthOffset.
  cellGcmAddressToOffset ( depthBuffer , &depthOffset );

  //Surfaces[] Contains the buffers that will be rendered into
  CellGcmSurface surfaces[BUFFERS_COUNT];

  for( int i = 0; i < BUFFERS_COUNT; ++i)
  {
    ///Allocate a 64byte aligned segment of RSX memory that is the size of a colour buffer
    void *buffer = LocalMemoryAlign (64 , color_size );

    //Get the offset address for it and store it in surfaces[i].colorOffset [0]
    cellGcmAddressToOffset (buffer , &surfaces[i]. colorOffset [0]);

    /* This function registers a buffer that outputs to a display.
    This is the point where the buffer is actually written to local memory.
    Parameters:
    cellGcmSetDisplayBuffer ( Buffer ID (0 - 7), memory offset, pitch - Horizontal byte width,
      width - Horizontal resolution (number of pixels), height - Vertical resolution(number of pixels)
    */
    cellGcmSetDisplayBuffer (i, surfaces[i].colorOffset[0], color_pitch, resolution.width, resolution.height );

  // Now we set other parameters on each CellGcmSurface object

  //whether 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
  //Up to 4 color buffers can be used on a CellGcmSurface, but we only use 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 (16−bit depth or 24−bit depth and 8−bit stencil)
  surfaces [i]. depthFormat = CELL_GCM_SURFACE_Z24S8;
  //whether to place the depth buffer, main memory or local memory.
  surfaces [i]. depthLocation = CELL_GCM_LOCATION_LOCAL;
  //The offset address to our depth buffer (We 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 = resolution.width ;
  surfaces [i]. height = resolution.height ;
  //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
cellGcmSetSurface (& surfaces [0]);
//Used to keep track of the surface currently being rendered to.
uint8_t swapValue = 0;

// *** #Load Shaders# *************************
printf(”Loading shaders\n”);

// This loader code is specific to this example,
//  as the compiled shaders are at the bottom of this file in a char[] array

//Fragment program
CGprogram programFS;
//Fragment microcode
void* ucodeFS;

uint32_t offsetFS;
{
  const unsigned int dataSize = sizeof( fs_basic );

  //Allocate some heap memory the size of the shader code
  char * data = ( char *)malloc (dataSize);
  //Copy the shader code into that memory location
  memcpy(data, fs_basic, dataSize);
  //Cast the copied code data to a CGprogram object
  programFS = ( CGprogram )( void *) fs_basic ;

  //Initialize the Cg binary program on memory for use by RSX.
  cellGcmCgInitProgram ( programFS );

  unsigned int ucodeSize ;
  void* ucodePtr;

  //Stores pointer to the microcode in ucodePtr, and the size of the microcode into ucodeSize.
  cellGcmCgGetUCode( programFS , &ucodePtr , &ucodeSize );

  //Reserve some local memory to store the fragment shader microcode
  ucodeFS = LocalMemoryAlign(64 , ucodeSize );
  //Copy the microcode into local memory
  memcpy (ucodeFS , ucodePtr , ucodeSize );

  //Get offset of the fragment microcode in local memory, stor into &offsetFS.
  cellGcmAddressToOffset (ucodeFS , &offsetFS );

  printf(”Fragment shader loaded\t Size: %i bytes\n”, ucodeSize);
}


//Vertex program
CGprogram programVS;
//vertex microcode
void* ucodeVS;

{
  unsigned int dataSize = sizeof(vs_basic);
  //Allocate some heap memory the size of the shader code
  char * data = ( char *)malloc ( dataSize );
  //Copy the shader code into that memory location
  memcpy(data, vs_basic, dataSize);
  //Cast the copied code data to a CGprogram object
  programVS = ( CGprogram )( void *) data ;

  //Initialize the Cg binary program on memory for use by RSX.
  cellGcmCgInitProgram ( programVS );

  unsigned int ucodeSize ;

  //The vertex program is left in main memory instead of being transferred to local memory,
  //  since it will be ultimately loaded into the command buffer anyway.

  //Stores pointer to the microcode in ucodePtr, and the size of the microcode into ucodeVS
  cellGcmCgGetUCode( programVS, &ucodeVS, &ucodeSize );

  printf(”Vertex shader loaded\t Size: %i bytes\n”, ucodeSize);
}

/*
  With all that pointer and memory juggling, let’s recap where we are now.
  programFS and programVS are CGprograms in main memory
  ucodeVS is a pointer to the vertex shader microcode in main memory
  offsetFS is an offset address pointer to the fragment shader microcode in local memory
*/

// Set the current shaders to use
cellGcmSetFragmentProgram ( programFS , offsetFS );
cellGcmSetVertexProgram ( programVS , ucodeVS );

// *** #Setup Shaders# *********************
printf(”Linking shader parameters\n”);

/*
  CGparameter − shader program parameters/uniforms.
  (int)ParameterResource − RSX hardware that will process the parameter.
*/

// Resolve position and colour parameters.
CGparameter position = cellGcmCgGetNamedParameter(programVS, ”position”);
DBG_ASSERT(position);
CGparameter color = cellGcmCgGetNamedParameter(programVS, ”color”);
DBG_ASSERT(color);
CGparameter mvp = cellGcmCgGetNamedParameter ( programVS , ”modelViewProj” );
```

```
DBG_ASSERT(mvp);

// Get the index of the vertex and colour attribute that will be set for the vertex shader
// These are used for cellGcmSetVertexDataArray();
int PositionIndex = cellGcmCgGetParameterResource(programVS, position) - CG_ATTR0;
DBG_ASSERT(PositionIndex>=0);

int ColorIndex = cellGcmCgGetParameterResource(programVS, color) - CG_ATTR0;
DBG_ASSERT(ColorIndex>=0);

// Either this or cellGcmSetFragmentProgram(program, offset) should be called when a parameter changes.
// This command is more efficient as it only changes the _parameters_ in memory, not the whole program.
cellGcmSetUpdateFragmentProgramParameter( offsetFS );

//Identiy matrix for our model View projection transform
Matrix4 mat = Matrix4::identity();
Matrix4 tempMatrix = transpose ( mat );
//Send mvp to vertex shader
cellGcmSetVertexProgramParameter (mvp, (float*)&tempMatrix);


//--- #Vertex DATA i.e. the actual triangles that we'll draw # -----------------------------------

printf("Making loads of triangles \n");
const int numVerts = 3; // for our simple triangle

//Reserve space for the vertex buffer in local memory
//Remember, this function doesn't actually save anything to memory.
stVertex* vertexBuffer = (stVertex*)LocalMemoryAlign(128, sizeof(stVertex)*numVerts);

// Could set the vertex data here once - however, we modify it on the fly within the update loop

uint32_t VertexBufferOffset;
//Get the offset address for our vertex buffer in local memory
err = cellGcmAddressToOffset((void*)vertexBuffer, &VertexBufferOffset);
DBG_ASSERT(err==CELL_OK);

// *** #Main Loop# *********************************

// We are all ready - Just keep looping and drawing
while ( true )
{
  DBG_HALT
  // *-1-* set viewport ----------------------------------------------------

  //The viewport is the mapping of coordinates to the pixels in the frame buffer.
  //The viewport could be smaller than the screen buffer, but not in this case.
  //These settings also define where the origin (0,0,0) is, in this case, the centre of the screen.

  uint16_t x = 0; // starting position of the viewport ( left of screen )
  uint16_t y = 0; // starting position of the viewport (top of screen )
  uint16_t w = resolution.width ; // Width of viewport
  uint16_t h = resolution.height ; // Height of viewport
  float fmin = 0.0f; // Minimum z value
  float fmax = 1.0f; // Maximum z value

  // Scale our NDC coordinates to the size of the screen
  float scale[4];
  scale [0] = w * 0.5f;
  scale [1] = h * -0.5f; // Flip y axis !
  scale [2] = (fmax - fmin ) * 0.5f;
  scale [3] = 0.0f;

  // Translate from a range starting from -1 to a range starting at 0
  float offset[4];
  offset [0] = x + scale [0];
  offset [1] = y + h * 0.5f;
  offset [2] = (fmax + fmin ) * 0.5f;
  offset [3] = 0.0f;

  // analogous to the glViewport function ... but with extra values !
```

```
cellGcmSetViewport (x, y, w, h, fmin , fmax , scale , offset );

// *-2-* Clear buffers -----------------------------
cellGcmSetColorMask ( CELL_GCM_COLOR_MASK_R |
        CELL_GCM_COLOR_MASK_G |
        CELL_GCM_COLOR_MASK_B |
        CELL_GCM_COLOR_MASK_A );

// *-3-* Setup Scene rendering parameters ---------------------

cellGcmSetDepthTestEnable ( CELL_GCM_TRUE );
//cellGcmSetDepthTestEnable ( CELL_GCM_FALSE );

cellGcmSetDepthFunc ( CELL_GCM_LESS );
//cellGcmSetDepthFunc(CELL_GCM_NEVER);

cellGcmSetCullFaceEnable( CELL_GCM_FALSE );

//cellGcmSetBlendEnable(CELL_GCM_FALSE);
cellGcmSetDepthTestEnable(CELL_GCM_TRUE);

cellGcmSetShadeMode(CELL_GCM_SMOOTH);

// *-4-* Clear Scene ------------------------------------

// This funky bit of code smoothly bends the screen clear color between
// red and blue so we know we are rendering to the screen!
static float count = 0;
count += 0.1f;
unsigned char r = ((int)count)%255;
unsigned char g = 32;
unsigned char b = (255-(int)count)%255;
cellGcmSetClearColor ((b <<0)|(g <<8)|(r <<16)|(255 <<24));

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 );

// *-5-* Set shader and draw vertices ------------------------------------------------

// Put vertice data in here - doing it here so we can
// change it on the fly within the update loop if we want (i.e., for animations)

// Triangle - 3 corner vertices -----
// Bottom left (red)
vertexBuffer[0].x  = -1;
vertexBuffer[0].y  = -1;
vertexBuffer[0].z  =  0;
vertexBuffer[0].rgba = 0xff0000ff;

// Top middle (green)
vertexBuffer[1].x  =  sin (count);
vertexBuffer[1].y  =  1;
vertexBuffer[1].z  =  cos (count);
vertexBuffer[1].rgba = 0x00ff00ff;

// Bottom right (blue)
vertexBuffer[2].x  =  1;
vertexBuffer[2].y  = -1;
vertexBuffer[2].z  =  0;
vertexBuffer[2].rgba = 0x0000ffff;
// ---------------------------------------------------

// ** Vertex Data
cellGcmSetVertexDataArray(  PositionIndex,
          0,
          sizeof(stVertex),
          3,
          CELL_GCM_VERTEX_F,
          CELL_GCM_LOCATION_LOCAL,
          VertexBufferOffset);

cellGcmSetVertexDataArray(  ColorIndex,
          0,
          sizeof(stVertex),
          4,
          CELL_GCM_VERTEX_UB,
          CELL_GCM_LOCATION_LOCAL,
          VertexBufferOffset + sizeof(float)*3 );
```

```


    // set polygon fill mode
    cellGcmSetDrawArrays(CELL_GCM_PRIMITIVE_TRIANGLES, 0, numVerts);

    // *-6-*  Finished Drawing  Swap buffers ------------------------------------

    //If a flip is still in progress, wait until the previous flip ends.
    while ( cellGcmGetFlipStatus ()!=0)
    {
      sys_timer_usleep (100);
    }

    cellGcmResetFlipStatus ();

    //Do the flip
    cellGcmSetFlip (( uint8_t ) swapValue );
    cellGcmFlush ();
    //Stop the RSX executing commands until flip is done.
    cellGcmSetWaitFlip ();

    swapValue = ! swapValue ;
    cellGcmSetSurface (& surfaces [ swapValue ]);
  }

  puts("Goodbye: Quitting!\n");
  return 0;

}// End main(..)


// ********************************



/*
void main
(
  float4 color_in      : COLOR,
  out float4 color_out : COLOR
)
{
  color_out = color_in;
}
*/
unsigned char fs_basic[208] = {
  0x00, 0x00, 0x1B, 0x5C, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0xD0, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x20,
  0x00, 0x00, 0x00, 0xA0, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0xC0, 0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x0A, 0xC5,
  0x00, 0x00, 0x10, 0x05, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x86, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x10, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x0A, 0xC5, 0x00, 0x00, 0x10, 0x05, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x95,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x8F, 0x00, 0x00, 0x10, 0x02, 0x00, 0x00, 0x00, 0x01,
  0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x43, 0x4F, 0x4C, 0x4F, 0x52, 0x00, 0x63, 0x6F, 0x6C, 0x6F, 0x72, 0x5F,
  0x69, 0x6E, 0x00, 0x43, 0x4F, 0x4C, 0x4F, 0x52, 0x00, 0x63, 0x6F, 0x6C, 0x6F, 0x72, 0x5F, 0x6F, 0x75, 0x74, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00, 0x02, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x3E, 0x01, 0x01, 0x00, 0xC8, 0x01, 0x1C, 0x9D,
  0xC8, 0x00, 0x00, 0x01, 0xC8, 0x00, 0x3F, 0xE1
};

/*
void main
(
  float4 position : POSITION,
  float4 color  : COLOR,

  uniform float4x4 modelViewProj,

  out float4 oPosition : POSITION,
  out float4 oColor    : COLOR
)
{
  oPosition = mul(ModelViewProjMatrix, position);
  oColor = color;
}
*/
unsigned char vs_basic[672] = {
  0x00, 0x00, 0x1B, 0x5B, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x02, 0xA0, 0x00, 0x00, 0x00, 0x09, 0x00, 0x00, 0x00, 0x20,
  0x00, 0x00, 0x02, 0x60, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x02, 0x80, 0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x08, 0x41,
  0x00, 0x00, 0x10, 0x05, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x01, 0xD9, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x01, 0xD0, 0x00, 0x00, 0x10, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x08, 0x44, 0x00, 0x00, 0x10, 0x05, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x01, 0xE8,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0xE2, 0x00, 0x00, 0x10, 0x01, 0x00, 0x00, 0x00, 0x01,
  0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x28, 0x00, 0x00, 0x0C, 0xB8, 0x00, 0x00, 0x10, 0x06,
  0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x01, 0xEE, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x10, 0x01, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x18,
  0x00, 0x00, 0x0C, 0xB8, 0x00, 0x00, 0x10, 0x06, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x01, 0xFC, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x01, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x0C, 0xB8, 0x00, 0x00, 0x10, 0x06, 0xFF, 0xFF, 0xFF, 0xFF,
  0x00, 0x00, 0x02, 0x0D, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x01,
  0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x0C, 0xB8,
  0x00, 0x00, 0x10, 0x06, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x02, 0x1E, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x01, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x0C, 0xB8, 0x00, 0x00, 0x10, 0x06, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x02, 0x2F,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x01, 0x00, 0x00, 0x00, 0x02,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x18, 0x00, 0x00, 0x08, 0xC3, 0x00, 0x00, 0x10, 0x05,
  0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x02, 0x49, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x40,
  0x00, 0x00, 0x10, 0x02, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x18,
  0x00, 0x00, 0x08, 0xC5, 0x00, 0x00, 0x10, 0x05, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x02, 0x59, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x53, 0x00, 0x00, 0x10, 0x02, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x01,
  0x00, 0x00, 0x00, 0x00, 0x50, 0x4F, 0x53, 0x49, 0x54, 0x49, 0x4F, 0x4E, 0x00, 0x70, 0x6F, 0x73, 0x69, 0x74, 0x69, 0x6F,
  0x6E, 0x00, 0x43, 0x4F, 0x4C, 0x4F, 0x52, 0x00, 0x63, 0x6F, 0x6C, 0x6F, 0x72, 0x00, 0x6D, 0x6F, 0x64, 0x65, 0x6C, 0x56,
  0x69, 0x65, 0x77, 0x50, 0x72, 0x6F, 0x6A, 0x00, 0x6D, 0x6F, 0x64, 0x65, 0x6C, 0x56, 0x69, 0x65, 0x77, 0x50, 0x72, 0x6F,
  0x6A, 0x5B, 0x30, 0x5D, 0x00, 0x6D, 0x6F, 0x64, 0x65, 0x6C, 0x56, 0x69, 0x65, 0x77, 0x50, 0x72, 0x6F, 0x6A, 0x5B, 0x31,
  0x5D, 0x00, 0x6D, 0x6F, 0x64, 0x65, 0x6C, 0x56, 0x69, 0x65, 0x77, 0x50, 0x72, 0x6F, 0x6A, 0x5B, 0x32, 0x5D, 0x00, 0x6D,
  0x6F, 0x64, 0x65, 0x6C, 0x56, 0x69, 0x65, 0x77, 0x50, 0x72, 0x6F, 0x6A, 0x5B, 0x33, 0x5D, 0x00, 0x50, 0x4F, 0x53, 0x49,
  0x54, 0x49, 0x4F, 0x4E, 0x00, 0x6F, 0x50, 0x6F, 0x73, 0x69, 0x74, 0x69, 0x6F, 0x6E, 0x00, 0x43, 0x4F, 0x4C, 0x4F, 0x52,
  0x00, 0x6F, 0x43, 0x6F, 0x6C, 0x6F, 0x72, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
  0x00, 0x00, 0x00, 0x09, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x40, 0x1F, 0x9C, 0x6C, 0x00, 0x40, 0x00, 0x0D, 0x81, 0x06, 0xC0, 0x83, 0x0x60, 0x41, 0xFF, 0x80, 0x40, 0x1F, 0x9C, 0x6C,
  0x00, 0x40, 0x03, 0x0D, 0x81, 0x06, 0xC0, 0x83, 0x60, 0x41, 0xFF, 0x85
};
```

5.3 Executable Output

You won't see any sexy graphics on screen when you run the graphics program. However, you'll see the screen gradually blend between blue and red (i.e., the background), while a triangles is drawn on the screen. The program will continue to run in the render while modifying the vertices on the fly (as shown in Listing 7). You can dissect the code - possibly modify the vertices so more triangles are drawn on the screen, a procedural shape is drawn (e.g., sphere, cube, or changing height terrain)

Congratulations You have successfully compiled and run your first graphical program on the PS3. You are now ready to move forwards and start compiling more complex programs and take advantage of sound, and the game-pad controller.

6. 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.

While the first few sections of the code will be reusable in future projects, the code dealing with vertex arrays and shaders is not extensible in it's current state. In the next Graphics tutorial (Basic Graphics Framework), this code will be expanded out into a system of classes that can be extended and built upon

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] James Leiterman. Vector games math processors (wordware game math library) (isbn:978-1556229213), 2011. 2

[2] Syd Logan. Cross-platform development in c++: Building mac os x, linux, and windows applications (isbn:978-0321246424), 2007. 2

[3] Matthew Scarpino. Programming the cell processor: For games, graphics, and computational proccessing (isbn: 978-0136008866), 2011. 2

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