Lesson 2
Playstation Vita Development

Introduction to Vita Graphics(GXM)

Sam Serrels and Benjamin Kenwright[1]*

Abstract
A beginners guide to getting started with graphical programming and developing on Sony's Playstation Vita. This article gives a brief introduction for students to initializing graphics buffers and displaying them on the screen.

Keywords
Sony, PSVita, PlayStation, Setup, Windows, SDK, Development, ELF, SELF, Programming, Visual Studio, Debugging

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

Contents

Introduction

About the Edinburgh Napier University Game Technology Playstation Vita 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 Development Kits (DevKits) and encourage enthusiastic students to design and build their own games and applications during their spare time.

Previous Tutorials This tutorial assumes you have read the tutorial on compiling and deploying applications to the PS Vita.

Additional Reading In addition to the lesson tutorials, we would recommend reading a number of books on game an cross-platform development and coding, such as, Vector Maths and Optimisation [?], and Cross-Platform Development in C++ [?].

1. PS Vita Graphics

1.1 The Hardware

The Screen Having a screen attached to the device means that there is higher degree of control over displaying an image when compared to an external screen like a TV. The biggest advantage is the reduction in latency, a TV has it's own internal processing that can get in the way of your image hitting the screen, in addition to the delay of decoding and encoding the data to send it over a cable. Although the system has control of the screen, no new functions are exposed for use to use, all we receive is the speed benefit. The system does have a very accurate reading on the refresh of the screen, so we also get more accurate VSync flips.
A further benefit to a built ins screen, is that the resolution and pixel ratio is known, and it won't change.

The Gpu The SPU in the Vita is a **SGX543MP4+**, and it's deal is **Tile Based Deferred Rendering**. The GPU is split into 4 independent cores, rendering an image is split between them, each core the further splits their share of the image into tiles.

Differed rendering Splitting an image into tiles is often used when calculating lighting in a differed rendering scenario. The image will have been drawn without any lighting, only textures and normals, every light source in the scene is then checked to see which tiles it is visible in, then the tiles with visible lights get rendered with a lighting pass. This is an efficient way to calculate lighting when there can be far too many lights to calculate the old way (ray tracing to every surface, while they are being drawn). The Vita GPU splits the image into tiles before *anything* is rendered, this is how it splits the work between the cores.

Optimisations The SGX543 isn't your ordinary desktop Graphics chip, the biggest difference is that it was designed with power usesage in mind. The Vita is a portable device, which means the power comes from a battery, if the GPU was designed to get the highest performance possible whatever the power cost, then the Vita would no longer be portable system. Efficiency is key, and the GPU has a larger bag of tricks to pull to be as efficient as possible.

Hidden surface removal Normally, when rendering a 3D scene, a depth buffer is used to handle the depth of objects. This can result in a lot of wasted rendering. If an object is rendered, it goes through the full shader pipeline and gets shaded with the fragment shader. If another object is rendered in front of the first

object, all the time spent rendering has been wasted.
The Vita GPU has a process for avoiding wasted rendering, called Hidden surface removal. This checks the depth of every fragment before rendering, so only the very topmost fragments are rendered and visible.

Render Order To check all the fragments against each other, before anything is rendered, you must *have* all the fragments. Instead of rendering each object one by one, all the traditional render calls are stored, and then when you would normally flip the buffers, you hand off all the saved render call to GXM witch does it's magic and then does the actual rendering.

Scenes The object that stores all the render calls is called a **"Scene"**. Scenes are the basis of many functions and techniques when rendering on the Vita. The primary benefit is that they can be precomputed and reused.

1.2 The Software

GXM, the graphics API The GXM library is the library that handles rendering, shaders and keeping the GPU happy. It is very similar to the GCM library used on the PS3 in terms of render calls when drawing a frame, but the initial set-up and the work done before and after each frame differ significantly.

Shaders Shaders are programmed in the Nvidia CG language, and compiled on the development PC with a custom SONY Cg compiler. In a traditional render set-up you would load these compiled shader files on the GPU, then tell the GPU/shader the format of any incoming data (Stride/frequency etc...), then send the data, either as an input or as a uniform.

Shader Patching This process changes for the PS Vita. As the GPU uses tile based rendering, to boost performance shaders have to be 'Patched' with the format of the data before they are sent to the tiles to do work. This means that if you have to render two objects, with the same shader, but each object has a slightly different layout of vertices, you would need to patch the shader twice. This isn't a big deal, as it's easy to implement around, but it's one of the many quirks of a tile based GPU.

2. Example Code

The following peice of code is the bare minumum needed to utput anythign on the screen. GXM is mostly being bypassed, as we only use for creating the buffers. Once the buffer are created we write to them directly from the CPU then manually call a swap command.
GXM will not render anything unless it has a shader loaded, it has no immediate mode. Rendering a simple triangle with GXM requires substantially more code than is written here, this example is just to show how to get the screen outputting *something*.

Listing 1. GXM.c

```
//Linked with libSceDbg_stub.a, libSceGxm_stub.a, libSceDisplay_stub.a
#include <string.h> //for Memset
#include <libdbg.h>
#include <kernel.h>
#include <display.h>
#include <gxm.h>
#include <math.h>

// native resolution
#define DISPLAY_WIDTH      960
#define DISPLAY_HEIGHT      544
#define DISPLAY_STRIDE_IN_PIXELS  1024

//libgxm color format to render to
#define COLOR_FORMAT SCE_GXM_COLOR_FORMAT_A8B8G8R8
#define PIXEL_FORMAT SCE_DISPLAY_PIXELFORMAT_A8B8G8R8

//The number of back buffers
#define BUFFER_COUNT  2

// Helper macro to align a value
#define ALIGN(x, a)   (((x) + ((a) - 1)) & ~((a) - 1))

/*This structure is serialized during sceGxmDisplayQueueAddEntry,
and is used to pass arbitrary data to the display callback function,called
from an internal thread once the back buffer is ready to be displayed.
In this example, we only need to pass the base address of the buffer.*/

typedef struct DisplayData
{
  void *address;
} DisplayData;

static void nullCallback(const void *callbackData){};

// Helper function to allocate memory and map it for the GPU
static void *graphicsAlloc(SceKernelMemBlockType type, uint32_t size, uint32_t alignment, uint32_t attribs, SceUID *uid);

// User main thread parameters
extern const char     sceUserMainThreadName[]  = "GXM_Basic";
extern const int      sceUserMainThreadPriority = SCE_KERNEL_DEFAULT_PRIORITY_USER;
extern const unsigned int sceUserMainThreadStackSize  = SCE_KERNEL_STACK_SIZE_DEFAULT_USER_MAIN;

// Define a 1MB heap for this program
unsigned int  sceLibcHeapSize = 1*1024*1024;

// Buffers holding the pixel data
void* displayBufferData[BUFFER_COUNT]
// Sync objects assigned to each buffer
SceGxmSyncObject* displayBufferSync[BUFFER_COUNT]

void init(){
  // set up parameters
  SceGxmInitializeParams params;
  memset(&params,0,sizeof(SceGxmInitializeParams));
  params.flags = 0;
  params.displayQueueCallback = nullCallback;
  params.displayQueueCallbackDataSize = sizeof(DisplayData);
  params.displayQueueMaxPendingCount=BUFFER_COUNT
  params.parameterBufferSize= SCE_GXM_DEFAULT_PARAMETER_BUFFER_SIZE;//16MB

  // Initialize
  int err = sceGxmInitialize(&params);
  SCE_DBG_ASSERT(err == SCE_OK);
}

void createBuffers(){

  // Set up rendering parameters
  SceGxmRenderTargetParams renderTargetParams;
  memset(&renderTargetParams,0,sizeof(SceGxmRenderTargetParams));
  renderTargetParams.flags        = 0;
  renderTargetParams.width        = DISPLAY_WIDTH;
  renderTargetParams.height       = DISPLAY_HEIGHT;
  renderTargetParams.scenesPerFrame   = 1;
  renderTargetParams.multisampleMode    = SCE_GXM_MULTISAMPLE_NONE;
  renderTargetParams.multisampleLocations = 0;
  renderTargetParams.driverMemBlock   = SCE_UID_INVALID_UID;

  // create the render target
```

```
SceGxmRenderTarget *renderTarget;
err=sceGxmCreateRenderTarget(&renderTargetParams,&renderTarget);
SCE_DBG_ASSERT(err == SCE_OK);

// allocate memory and sync objects for display buffers
SceUID displayBufferUid[BUFFER_COUNT];
SceGxmColorSurface displaySurface[BUFFER_COUNT];

for (uint32_t i = 0; i < BUFFER_COUNT; ++i) {
 // allocate memory for display
 displayBufferData[i] = graphicsAlloc(
  SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RWDATA,
  4*DISPLAY_STRIDE_IN_PIXELS*DISPLAY_HEIGHT,
  SCE_GXM_COLOR_SURFACE_ALIGNMENT,
  SCE_GXM_MEMORY_ATTRIB_READ | SCE_GXM_MEMORY_ATTRIB_WRITE,
  &displayBufferUid[i]);

 // initialize a color surface for this display buffer
 err = sceGxmColorSurfaceInit(
  &displaySurface[i],
  COLOR_FORMAT,
  SCE_GXM_COLOR_SURFACE_LINEAR,
  SCE_GXM_COLOR_SURFACE_SCALE_NONE ,
  SCE_GXM_OUTPUT_REGISTER_SIZE_32BIT,
  DISPLAY_WIDTH,
  DISPLAY_HEIGHT,
  DISPLAY_STRIDE_IN_PIXELS,
  displayBufferData[i]);
 SCE_DBG_ASSERT(err == SCE_OK);

 // create a sync object that we will associate with this buffer
 err = sceGxmSyncObjectCreate(&displayBufferSync[i]);
 SCE_DBG_ASSERT(err == SCE_OK);
}
}

// Entry point
int main(void)
{
 init();
 createBuffers();

 int err = 0;
 bool flip = false;
 float count = 0.0f;

 SceDisplayFrameBuf framebuf;
 framebuf.size        = sizeof(SceDisplayFrameBuf);
 framebuf.pitch       = DISPLAY_STRIDE_IN_PIXELS;
 framebuf.pixelformat = PIXEL_FORMAT;
 framebuf.width       = DISPLAY_WIDTH;
 framebuf.height      = DISPLAY_HEIGHT;

 while ( true )
 {
  flip = !flip;
  count += 0.1f;
  // Smooth colour cycle
  unsigned char r = (sin((0.1f*count) + 0) * 127) + 128;
  unsigned char g = (sin((0.1f*count) + 2) * 127) + 128;
  unsigned char b = (sin((0.1f*count) + 4) * 127) + 128;
  int colour = (b << 0) | (g << 8) | (r << 16) | (255 << 24);
  int h = ((((int)(10.0f*count) % (DISPLAY_WIDTH)) + (DISPLAY_WIDTH)) % (DISPLAY_WIDTH));

  //Write color data to displayBufferData
  for (uint32_t y = 0; y < DISPLAY_HEIGHT; ++y) {
   uint32_t *row = (uint32_t *)displayBufferData[(int)flip] + y*DISPLAY_STRIDE_IN_PIXELS;
   for (uint32_t x = h; x < DISPLAY_WIDTH; ++x)
   {
    row[x] = colour;
   }
  }

  framebuf.base = displayBufferData[(int)flip];

  // Swap to the new buffer on the next VSYNC
  err = sceDisplaySetFrameBuf(&framebuf, SCE_DISPLAY_UPDATETIMING_NEXTVSYNC);
  SCE_DBG_ASSERT(err == SCE_OK);
  // Block this callback until the swap has finished
  err = sceDisplayWaitVblankStart();
  SCE_DBG_ASSERT(err == SCE_OK);
 }
}

//! Alocates memory either on CDRAM or LPDDR, maps it for GPU
static void *graphicsAlloc(SceKernelMemBlockType type, uint32_t size, uint32_t alignment, uint32_t attribs, SceUID *uid)
{/*
 TheKernelAllocMemBlock func doesn't use an alignment parameter
 So we must calcualte an aligned size large enough to accommodate
 whatever we need to store. If you were using your own custom func
 for allcoation, you could use the alignment parameter. The minimum
 alignment size is different for each type of memory. */
 if (type == SCE_KERNEL_MEMBLOCK_TYPE_USER_CDRAM_RWDATA) {
  // CDRAM memblocks must be 256KB aligned
  SCE_DBG_ASSERT(alignment <= 256*1024);
  size = ALIGN(size, 256*1024);
 } else {
  // LPDDR memblocks must be 4KB aligned
  SCE_DBG_ASSERT(alignment <= 4*1024);
  size = ALIGN(size, 4*1024);
 }

 // allocate the memeory
 *uid = sceKernelAllocMemBlock("basic", type, size, NULL);
 SCE_DBG_ASSERT(*uid >= SCE_OK);

 // grab the base address
 void *baseAddr = NULL;
 int err = sceKernelGetMemBlockBase(*uid, &baseAddr);
 SCE_DBG_ASSERT(err == SCE_OK);

 // Map for the GPU (Attribs = Read/write permissions)
 err = sceGxmMapMemory(baseAddr, size, attribs);
 SCE_DBG_ASSERT(err == SCE_OK);

 return baseAddr;
}
```

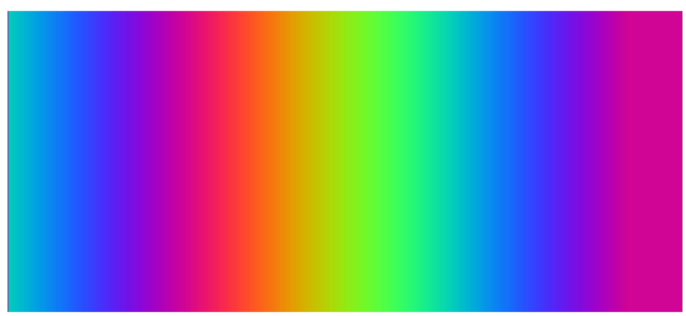

Figure 1. Screen Output - You should see a rainbow pattern swiping from left to right

Output The program should deploy quickly to the Devkit, execute, then close. To catch the output before it closes, open the "Console output for Playstation Vita" program from the start menu or from Neighbourhood. (See figure **??**)

3. Conclusion

Rendering is very large topic, and there are plenty more interesting features unique to the Vita GPU, this has been a very brief introduction to the topic.

Acknowledgements

The lessons provide a basic introduction for getting started with Sony's Playstation Vita console development. So if you can provide any advice, tips, or hints during from your own exploration of PSVita development, that you think would be indispensable for a student's learning and understanding, please don't hesitate to contact us so that we can make amendments and incorporate them into future tutorials.

Recommended Reading

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