The Vulkan Sample Code Included with These Notes

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

The Vulkan Sample Code Included with These Notes

Sample Program Keyboard Inputs

Y, T: Toggle using a vertex buffer only vs. an index buffer
T, L: Toggle lighting off and on
M, I: Toggle display mode (textures vs. colors, for now)
P, P: Pause the animation
O, Q: Quit the program
Esc: Quit the program
Y, R: Toggle rotation-animation and using the mouse

Sample Program Output

Caveats on the Sample Code, I

1. I've written everything out in appalling longhand.
2. Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything easily.
3. At times, I could have hidden complexity, but I didn't. At all stages, I have tried to err on the side of showing you everything, so that nothing happens in a way that's kept a secret from you.
4. I've setup Vulkan structs every time they are used, even though, in many cases, they could have been setup once and then re-used each time.
5. At times, I've setup things that didn't need to be setup just to show you what could go there.

Caveats on the Sample Code, II

6. There are good uses for C++ classes and methods here to hide some complexity, but I've not done that.
7. I've typedef'd a couple things to make the Vulkan phraseology more consistent.
8. Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure.
9. At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
10. I've divided functionality up into the pieces that make sense to me. Many other divisions are possible. Feel free to invent your own.

Main Program

```cpp
int main( int argc, char * argv[] )
{
    Width  = 800;
    Height = 600;
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open debug print file '%s'
", DEBUGFILE );
        FpDebug = stderr;
    }
    fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d
", Width, Height);
    Reset( );
    InitGraphics( );
    // loop until the user closes the window:
    while( glfwWindowShouldClose( MainWindow ) == 0 )
    {
        glfwPollEvents( );
        Time = glfwGetTime();          // elapsed time, in double-precision seconds
        UpdateScene( );
        RenderScene( );
    }
    fprintf(FpDebug, "Closing the GLFW window
");
    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
```
```c
void InitGraphics()
{
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance();
    InitGLFW();
    Init02CreateDebugCallbacks();
    Init03PhysicalDeviceAndGetQueueFamilyProperties();
    Init04LogicalDeviceAndQueue();
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer);
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );
    Init06CommandPool();
    Init06CommandBuffers();
    Init07TextureSampler( &MyPuppyTexture.texSampler );
    Init07TextureBufferAndFillFromBmpFile( "puppy.bmp", &MyPuppyTexture );
    Init08Swapchain();
    Init09DepthStencilImage();
    Init10RenderPasses();
    Init11Framebuffers();
    Init12SpirvShader( "sample-vert.spv", &ShaderModuleVertex );
    Init12SpirvShader( "sample-frag.spv", &ShaderModuleFragment );
    Init13DescriptorSetPool();
    Init13DescriptorSetLayouts();
    Init13DescriptorSets();
    Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment,
            VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );

    static GLuint CubeTriangleIndices[ ][3] = {
        { 0, 2, 3 },
        { 0, 3, 1 },
        { 4, 5, 7 },
        { 4, 7, 6 },
        { 1, 3, 7 },
        { 1, 7, 5 },
        { 0, 4, 6 },
        { 0, 6, 2 },
        { 2, 6, 7 },
        { 2, 7, 3 },
        { 0, 1, 5 },
        { 0, 5, 4 };

    struct vertex VertexData[ ] = {
        // triangle 0-2-3:
        // vertex #0:
        { -1., -1., -1. },
        {  0.,  0., -1. },
        {  0.,  0.,  0. },
        {  1., 0. },
        // vertex #2:
        { -1.,  1., -1. },
        {  0.,  0., -1. },
        {  0.,  1.,  0. },
        {  1., 1. },
        // vertex #3:
        {  1.,  1., -1. },
        {  0.,  0., -1. },
        {  1.,  1.,  0. },
        {  0., 1. },
    };

    #include "SampleVertexData.cpp"

    struct vertex
    {
        glm::vec3        position;
        glm::vec3        normal;
        glm::vec3        color;
        glm::vec2        texCoord;
    };

    struct vertex VertexData[ ] = {
        // triangle 0-2-3:
        // vertex #0:
        { -1., -1., -1. },
        {  0.,  0., -1. },
        {  0.,  0.,  0. },
        {  1., 0. },
        // vertex #2:
        { -1.,  1., -1. },
        {  0.,  0., -1. },
        {  0.,  1.,  0. },
        {  1., 1. },
        // vertex #3:
        {  1.,  1., -1. },
        {  0.,  0., -1. },
        {  1.,  1.,  0. },
        {  0., 1. },
    };

    . . .
```

---

**The Vertex Data is in a Separate File**

```c
#include "SampleVertexData.cpp"
```

```c
struct vertex
{
    glm::vec3        position;
    glm::vec3        normal;
    glm::vec3        color;
    glm::vec2        texCoord;
};
```

For example, what if you are not doing texturing in this application? Should you re-do this struct and leave the texCoord element out?

As best as I can tell, the only penalties for leaving in vertex attributes that you aren’t going to use is memory space and possibly some inefficient uses of the cache, but not gross performance. So, I recommend keeping this struct intact, and, if you don’t need texturing, simply don’t use the texCoord values in your vertex shader.
Vulkan Software Philosophy

1. There are lots of typedefs that define C/C++ structs and enums.

2. Vulkan takes a non-C++ object-oriented approach in that those typedefed structs pass all the necessary information into a function. For example, where we might normally say in C++:

   ```cpp
   result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex, OUT &Queue );
   ```

   we would actually say in C:

   ```c
   result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
   ```

Vulkan Conventions

VKxxx is a typedef, probably a struct

vkxxx( ) is a function call

VK_XXX is a constant

My Conventions

"Init" in a function call name means that something is being setup that only needs to be setup once.

The number after "Init" gives you the ordering.

In the source code, after main() comes InitGraphics(), then all of the InitXXX() functions in numerical order. After that comes the helper functions.

"Find" in a function call name means that something is being looked for.

"Fill" in a function call name means that some data is being supplied to Vulkan.

"IN" and "OUT" ahead of function call arguments are just there to let you know how an argument is going to be used by the function. Otherwise, IN and OUT have no significance. They are each actually #define'd to nothing.

Querying the Number of Something and Allocating Enough Structures to Hold Them All

```c
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr );
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ count ];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT &physicalDevices[0] );
result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, &physicalDevices[0] );
```
Extras in the Code

```c
#define REPORT(s)               { PrintVkError( result, s );  fflush(FpDebug); }
#define HERE_I_AM(s)          if( Verbose )  { fprintf( FpDebug, "***** %s *****\n", s );  fflush(FpDebug); }

bool Paused;
bool Verbose;

#define DEBUGFILE              "VulkanDebug.log"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
```