Vulkan Sample Code

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

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

Vulkan Sample Code

Caveats on the Sample Code

• I’ve written everything out in appalling longhand.
• Everything is in one .cpp file (except the geometry data). It really should be broken up, but this way you can find everything.
• 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 a secret to you.
• 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.
• At times, I’ve setup things that didn’t need to be setup just to show you what could go there.
• There are good uses for C++ classes and methods here to hide some complexity, but I’ve not done that.
• I’ve typedef’ed a couple things to make the Vulkan phraseology more consistent.
• Even though it is not good software style, I have put persistent information in global variables, rather than a separate data structure
• At times, I have copied lines from vulkan.h into the code as comments to show you what certain options could be.
• 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

```c
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'
" );
        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;
}
```

Main Program Loop
void InitGraphics() {
    HERE_I_AM( "InitGraphics" );
    VkResult result = VK_SUCCESS;
    Init01Instance();
    Init02LogicalDeviceAndQueue();
    Init03PhysicalDeviceAndGetQueueFamilyProperties();
    Init04LogicalDeviceAndQueue();
    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );
    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 );
}

A Colored Cube

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. },

    // vertex #1:
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1., 1. },

    // vertex #4:
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. },

    // vertex #5:
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. },
The Vertex Data is in a Separate File

```cpp
#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. }
};
```

What if you don't need all of this information?

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 penalty for leaving in vertex attributes you aren't going to use is memory space, but not 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 InitxxYYY() 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 pointer (address) arguments are just there to let you know how a pointer is used by the function. Otherwise, they have no significance.

```c
#define IN
#define OUT
```
uint32_t count;
result = vkEnumeratePhysicalDevices(Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr);

VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices(Instance, OUT &count, physicalDevices);

Where to put them

How many total there are

This way of querying information is a recurring OpenCL and Vulkan pattern (get used to it):

void PrintVkError( VkResult result, std::string prefix )
{
    if (Verbose && result == VK_SUCCESS)
    {
        fprintf(FpDebug, "%s: %s
", prefix.c_str(), "Successful");
        fflush(FpDebug);
        return;
    }
    const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
    std::string meaning = "";
    for( int i = 0; i < numErrorCodes; i++ )
    {
        if( result == ErrorCodes[i].resultCode )
        {
            meaning = ErrorCodes[i].meaning;
            break;
        }
    }
    fprintf(FpDebug, "%s: %s
", prefix.c_str(), meaning.c_str());
    fflush(FpDebug);
}
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     Paued;
bool     Verbose;

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