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

Mike Bailey
mjb@cs.oregonstate.edu

Sample Program Keyboard Inputs

'Y', 'y': Toggle lighting off and on
'Y', 'Y': Toggle display mode (textures vs. colors, for now)
'P', 'p': Pause the animation
'O', 'O': quit the program
Esc: quit the program
'Y', 'y': Toggle rotation-animation and using the mouse
'Y', 'Y': Toggle using a vertex buffer only vs. an index buffer (in the index buffer version)
'1', '4', '9': Set the number of instances (in the instancing version)

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 (most?), 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 great uses for C++ classes and methods here to hide some complexity, but I’ve not done that.
7. I’ve typedefed 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

```
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: Width = %d ; Height = %d

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();
    Init2CreateDebugCallback();
    Init2PhysicalDeviceAndQueueFamilyProperties();
    Init3UniformBuffer(sizeof(Matrices), &MyMatrixUniformBuffer);
    Fill05DataBuffer(MyMatrixUniformBuffer, (void *)&Matrices);
    Init3UniformBuffer(sizeof(Light), &MyLightUniformBuffer);
    Fill05DataBuffer(MyLightUniformBuffer, (void *)&Light);
    Init3MyVertexDataBuffer(sizeof(VertexData), &MyVertexDataBuffer);
    Fill05DataBuffer(MyVertexDataBuffer, (void *)&VertexData);
    Init4CommandPool();
    Init4CommandBuffers();
    Init7TextureSampler(&MyPuppyTexture.texSampler);
    Init7TextureBufferAndFillFromBmpFile("puppy.bmp", &MyPuppyTexture);
    Init9Swapchain();
    Init9DepthStencilImage();
    Init10RenderPasses();
    Init11Framebuffers();
    Init12SpirvShader("sample-vert.spv", &ShaderModuleVertex);
    Init12SpirvShader("sample-frag.spv", &ShaderModuleFragment);
    Init13DescriptorSetPool();
    Init13DescriptorSetLayouts();
    Init13DescriptorSets();
    Init14GraphicsVertexFragmentPipeline(ShaderModuleVertex, ShaderModuleFragment,
                                              VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline);
}
```

```c
static GLuint CubeTriangleIndices[8] = {
    { 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 }
};
```

```c
struct vertex VertexData[8] = {
    // 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.,  0.,  1. },
        {  1., 1. }
    },
    // vertex #4:
    {
        { -1., -1.,  1. },
        {  0.,  0.,  1. },
        {  0.,  0.,  0. },
        {  1., 1. }
    },
    // vertex #5:
    {
        {  1.,  1.,  0. },
        {  0.,  0.,  0. },
        {  1.,  1.,  0. },
        {  0., 1. }
    },
    // vertex #6:
    {
        {  1.,  1.,  0. },
        {  0.,  0.,  0. },
        {  1.,  1.,  0. },
        {  0., 1. }
    },
    // vertex #7:
    {
        {  1., -1.,  0. },
        {  0.,  0.,  0. },
        {  1., -1.,  0. },
        {  0., -1. }
    }
};
```

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

```c
struct vertex VertexData[8] = {
    // 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.,  0.,  1. },
        {  1., 1. }
    },
    // vertex #4:
    {
        { -1., -1.,  1. },
        {  0.,  0.,  1. },
        {  0.,  0.,  0. },
        {  1., 1. }
    },
    // vertex #5:
    {
        {  1.,  1.,  0. },
        {  0.,  0.,  0. },
        {  1.,  1.,  0. },
        {  0., 1. }
    },
    // vertex #6:
    {
        {  1.,  1.,  0. },
        {  0.,  0.,  0. },
        {  1.,  1.,  0. },
        {  0., 1. }
    },
    // vertex #7:
    {
        {  1., -1.,  0. },
        {  0.,  0.,  0. },
        {  1., -1.,  0. },
        {  0., -1. }
    }
};
```

The Vertex Data is in a Separate File

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 costs for retaining vertex attributes that you aren't going to use are some GPU 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

Vulkan has lots of typedefs that define C/C++ structs and enums. Vulkan takes a non-C++ object-oriented approach in that those typedef'ed 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

`vkYyy()` is a function call

`VK_ZZZ` is a constant

My Conventions

- **Vk**Xxx is a typedef, probably a struct
- **vk**Yyy() is a function call
- **VK**_ZZZ is a constant

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 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] );
```

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

- How many total
- How many to put
- Where to put them

Reporting Error Results, I

```c
struct errorcode {
    VkResult resultCode;
    std::string meaning;
};
ErrorCodes[] = {
    { VK_NOT_READY, "Not Ready" },
    { VK_TIMEOUT, "Timeout" },
    { VK_EVENT_SET, "Event Set" },
    { VK_EVENT_RESET, "Event Reset" },
    { VK_INCOMPLETE, "Incomplete" },
    { VK_ERROR_OUT_OF_HOST_MEMORY, "Out of Host Memory" },
    { VK_ERROR_OUT_OF_DEVICE_MEMORY, "Out of Device Memory" },
    { VK_ERROR_INITIALIZATION_FAILED, "Initialization Failed" },
    { VK_ERROR_DEVICE_LOST, "Device Lost" },
    { VK_ERROR_MEMORY_MAP_FAILED, "Memory Map Failed" },
    { VK_ERROR_LAYER_NOT_PRESENT, "Layer Not Present" },
    { VK_ERROR_EXTENSION_NOT_PRESENT, "Extension Not Present" },
    { VK_ERROR_FEATURE_NOT_PRESENT, "Feature Not Present" },
    { VK_ERROR_INCOMPATIBLE_DRIVER, "Incompatible Driver" },
    { VK_ERROR_TOO_MANY_OBJECTS, "Too Many Objects" },
    { VK_ERROR_FORMAT_NOT_SUPPORTED, "Format Not Supported" },
    { VK_ERROR_FRAGMENTED_POOL, "Fragmented Pool" },
    { VK_ERROR_SURFACE_LOST_KHR, "Surface Lost" },
    { VK_ERROR_NATIVE_WINDOW_IN_USE_KHR, "Native Window In Use" },
    { VK_ERROR_RECOGNIZED_ERROR_KHR, "Recognized Error" },
    { VK_ERROR_UNSUPPORTED_OPTION_KHR, "Unsupported Option" },
    { VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" },
    { VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory" },
    { VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" },
    { VK_ERROR_INVALID_SHADER_NV, "Invalid Shader" },
    { VK_ERROR_OUT_OF_HOST_MEMORY_KHR, "Out of Host Memory" },
    { VK_ERROR_OUT_OF_DEVICE_MEMORY_KHR, "Out of Device Memory" },
};
```

Reporting Error Results, II

```c
void PrintVkError( VkResult result, std::string prefix ) {
    if (Verbose && result == VK_SUCCESS) {
        fprintf(FpDebug, "%s: %s\n", 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\n", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}
```
Extras in the Code

#define REPORT(s)               { PrintVkError( result, s );  fflush(FpDebug); }
#define HERE_I_AM(s)          if( Verbose )  { fprintf( FpDebug, "***** %s *****
", s );  fflush(FpDebug); }
bool Paused;
bool Verbose;
#define DEBUGFILE               "VulkanDebug.txt"
errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
offsetof( uint32_t, OFFSET ZERO ) = 0;