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 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'd 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'
", 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;
}
```

InitGraphics()}
A Colored Cube

```c
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 }
};
```

A Colored Cube

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

```c
struct vertex VertexData[3] = {
  // 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"
```

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

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

```c
struct vertex VertexData[3] = {
  // 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. } },

  ... 
};
```

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++:
   ```c
   result = LogicalDevice->vkGetDeviceQueue ( queueFamilyIndex, queueIndex,  OUT &Queue );
   ```
   we would actually say in C:
   ```c
   result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
   ```

Vulkan Conventions

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, they have no significance.
```c
uint32_t count;
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT (VkPhysicalDevice *)nullptr);
VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[count];
result = vkEnumeratePhysicalDevices( Instance, OUT &count, OUT physicalDevices);
```

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

```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_SUBOPTIMAL_KHR, "Suboptimal"},
    {VK_ERROR_OUT_OF_DATE_KHR, "Error Out of Date"},
    {VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display"},
    {VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed"},
    {VK_ERROR_INVALID_SHADER_NV, "Invalid Shader"},
    {VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory"},
    {VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR, "Invalid External Handle"},
};
```

```c
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, "\n%s: %s\n", prefix.c_str(), meaning.c_str() );
    fflush(FpDebug);
}
```

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

Extras in the Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```