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'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.
- 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\"\n", DEBUGFILE);
    FpDebug = stderr;
  }
  fprintf(FpDebug, "FpDebug: Width = %d ; Height = %d\n", 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\n");
  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();
Init12Shader("sample-vert.spv", &ShaderModuleVertex );
Init12Shader("sample-frag.spv", &ShaderModuleFragment );
Init13DescriptorSetPool();
Init13DescriptorSetLayouts();
Init13DescriptorSets();
Init14GraphicsVertexFragmentPipeline( ShaderModuleVertex, ShaderModuleFragment, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```

```
struct vertex
{
  glm::vec3       position;
  glm::vec3       normal;
  glm::vec3       color;
  glm::vec2       texCoord;
};
```
The Vertex Data is in a Separate File

The Vertex Data is in a Separate File

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

The Vertex Data is in a Separate File

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

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 typedef'd 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

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 typedef’d 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

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 typedef’d 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

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 typedef’d 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

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 typedef’d 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);
```
Querying the Number of Something and Allocating Structures to Hold Them All

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): 

How many total there are Where to put them

result = vkEnumeratePhysicalDevices( Instance, &count, nullptr );
result = vkEnumeratePhysicalDevices( Instance, &count, physicalDevices );

Reporting Error Results, I

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, "Out of Date" }, 
{ VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" }, 
{ VK_ERROR_VALIDATION_FAILED_EXT, "Validation Failed" }, 
{ VK_ERROR_EXTERNAL_MEMORY_INUSE_KHR, "External Memory InUse" }, 
{ VK_ERROR_INCOMPLETE_KHR, "Incomplete" }, 
{ VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory" }, 
{ VK_ERROR_INCOMPATIBLE_DISPLAY_KHR, "Incompatible Display" }, 
{ VK_ERROR_INVALID_EXTERNAL_HANDLE_KHR, "Invalid External Handle" } ,

Reporting Error Results, II

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);
}
#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" );