

```

// *****
// This is a program for teaching Vulkan
// As such, it is deliberately verbose so that it is obvious (as possible, at least) what is being done
//
// Mike Bailey, Oregon State University
// mjb@cs.oregonstate.edu
//
// The class notes for which this program was written can be found here:
// http://cs.oregonstate.edu/~mjb/vulkan
//
// Keyboard commands:
// 'i', 'I': Toggle using a vertex buffer only vs. a vertex/index buffer
// 'l', 'L': Toggle lighting off and on
// 'm', 'M': Toggle display mode (textures vs. colors, for now)
// 'p', 'P': Pause the animation
// 'q', 'Q': Esc: exit the program
// 'r', 'R': Toggle rotation-animation and using the mouse
// '1', '4', '9'   Number of instances
//
// This code occasionally uses #defines for environment-specific-isms:
// _WIN32           Windows
// _linux_         Linux
// __GNUC__       GNU compiler
//
// There are some pieces of the program that were tossed in to show how to do some things
// they are not necessary in the program
// #define them to turn them on, #undef them to turn them off
// #undef EXAMPLE_OF_USING_DYNAMIC_STATE_VARIABLES
//
// There are also some spots where options are listed just to show you what could have happened
// here:
//     VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
//     #ifdef CHOICES
//     VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
//     VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
//     . . .
//     #endif
//
//
// Latest update: March 1, 2018
// *****
// *****

// *****
// INCLUDES:
// *****

#ifdef _WIN32
#include <io.h>
#endif
#include <stdlib.h>
#include <stdio.h>
// #include <unistd.h>
#include <math.h>
#ifndef M_PI
#define M_PI 3.14159265f
#endif

#include <stdarg.h>
#include <string.h>
#include <string>
#include <stdbool.h>
#include <assert.h>
#include <signal.h>

#include <vector>

#ifndef _WIN32
typedef int errno_t;

```

```

int      fopen_s( FILE**, const char *, const char * );
#endif

#define GLFW_INCLUDE_VULKAN
#include "glfw3.h"

#define GLM_FORCE_RADIANS
// #define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include "glm/vec2.hpp"
#include "glm/vec3.hpp"
#include "glm/mat4x4.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/matrix_inverse.hpp"
// #include "glm/gtc/type_ptr.hpp"

#ifdef _WIN32
// #pragma comment(linker, "/subsystem:windows")
#define APP_NAME_STR_LEN 80
#endif

#include "vulkan.h"
#include "vk_sdk_platform.h"

// these are here to flag why addresses are being passed into a vulkan function --
// 1. is it because the function wants to consume the contents of that structure or array (IN)?
// or, 2. is it because that function is going to fill that structure or array (OUT)?
#define IN
#define OUT
#define INOUT

// *****
// DEFINED CONSTANTS:
// *****

// useful stuff:

#define DEBUGFILE           "VulkanDebug.txt"
#define nullptr            (void *)NULL
#define MILLION             1000000L
#define BILLION             1000000000L
#define TEXTURE_COUNT      1
#define APP_SHORT_NAME     "Cube Sample"
#define APP_LONG_NAME      "Vulkan Cube Sample Program"

#define SECONDS_PER_CYCLE  3.f
#define FRAME_LAG          2
#define SWAPCHAINIMAGECOUNT 2

// multiplication factors for input interaction:
// // (these are known from previous experience)

const float ANGFACT = { M_PI/180.f };
const float SCLFACT = { 0.005f };

// minimum allowable scale factor:

const float MINSCALE = { 0.05f };

// active mouse buttons (or them together):

const int LEFT   = { 4 };
const int MIDDLE = { 2 };
const int RIGHT  = { 1 };

// the allocation callbacks could look like this:
// typedef struct VkAllocationCallbacks {
// void* pUserData;

```

```

//PFN_vkAllocationFunction          pfnAllocation;
//PFN_vkReallocationFunction        pfnReallocation;
//PFN_vkFreeFunction                pfnFree;
//PFN_vkInternalAllocationNotification pfnInternalAllocation;
//PFN_vkInternalFreeNotification    pfnInternalFree;
//} VkAllocationCallbacks;
// but we are not going to use them for now:
#define PALLOCATOR (VkAllocationCallbacks *)nullptr

// report on a result return:

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

// graphics parameters:

const double FOV =          glm::radians(60.);          // field-of-view angle
const float EYEDIST =      3.;                          // eye distance
const float OMEGA =        2.*M_PI;                    // angular velocity, radians/sec

#define SPIRV_MAGIC        0x07230203
// if you do an od -x, the magic number looks like this:
// 0000000 0203 0723 . . .

#define NUM_QUEUES_WANTED  1

#define ARRAY_SIZE(a)      (sizeof(a) / sizeof(a[0]))

// these are here for convenience and readability:
#define VK_FORMAT_VEC4      VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_XYZW     VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_VEC3     VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_STP      VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_XYZ      VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_VEC2     VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_ST       VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_XY       VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_FLOAT    VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_S        VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_X        VK_FORMAT_R32_SFLOAT

// my own error codes:
#define VK_FAILURE          (VkResult)( -2000000000 )
#define VK_SHOULD_EXIT     (VkResult)( -2000000001 )
#define VK_ERROR_SOMETHING_ELSE (VkResult)( -2000000002 )

```

```
// *****
// MY HELPER TYPEDEFS AND STRUCTS FOR VULKAN WORK:
// *****

typedef VkBuffer          VkDataBuffer;
typedef VkDevice         VkLogicalDevice;
typedef VkDeviceCreateInfo VkLogicalDeviceCreateInfo;
#define vkCreateLogicalDevice vkCreateDevice

// holds all the information about a data buffer so it can be encapsulated in one variable:
typedef struct MyBuffer
{
    VkDataBuffer          buffer;
    VkDeviceMemory       vdm;
    VkDeviceSize         size;
} MyBuffer;

typedef struct MyTexture
{
    uint32_t              width;
    uint32_t              height;
    unsigned char *       pixels;
    VkImage               texImage;
    VkImageView           texImageView;
    VkSampler             texSampler;
    VkDeviceMemory       vdm;
} MyTexture;

// bmp file headers:
struct bmfh
{
    short bfType;
    int bfSize;
    short bfReserved1;
    short bfReserved2;
    int bfOffBits;
} FileHeader;

struct bmih
{
    int biSize;
    int biWidth;
    int biHeight;
    short biPlanes;
    short biBitCount;
    int biCompression;
    int biSizeImage;
    int biXPelsPerMeter;
    int biYPelsPerMeter;
    int biClrUsed;
    int biClrImportant;
} InfoHeader;

// *****
// STRUCTS FOR THIS APPLICATION:
// *****

// uniform variable block:
struct matBuf
{
    glm::mat4 uModelMatrix;
    glm::mat4 uViewMatrix;
    glm::mat4 uProjectionMatrix;
}
```

```
        glm::mat4 uNormalMatrix;
};

// uniform variable block:
struct lightBuf
{
    float uKa;
    float uKd;
    float uKs;
    float uShininess;
    glm::vec4 uLightPos;
    glm::vec4 uLightSpecularColor;
    glm::vec4 uEyePos;
};

// uniform variable block:
struct miscBuf
{
    float uTime;
    int    uMode;
    int    uLighting;
    int    uNumInstances;
};

// an array of this struct will hold all vertex information:
struct vertex
{
    glm::vec3    position;
    glm::vec3    normal;
    glm::vec3    color;
    glm::vec2    texCoord;
};
```

```

// *****
// VULKAN-RELATED GLOBAL VARIABLES:
// *****

VkCommandBuffer          CommandBuffers[2];           // 2, because of double-
buffering
VkPipeline               ComputePipeline;
VkPipelineCache          ComputePipelineCache;
VkPipelineLayout        ComputePipelineLayout;
VkDataBuffer            DataBuffer;
VkImage                 DepthStencilImage;
VkImageView             DepthStencilImageView;
VkDescriptorPool        DescriptorPool;
VkDescriptorSetLayout   DescriptorSetLayouts[4];
VkDescriptorSet         DescriptorSets[4];
VkDebugReportCallbackEXT ErrorCallback = VK_NULL_HANDLE;
VkEvent                Event;
VkFence               Fence;
VkFramebuffer        Framebuffers[2];
VkCommandPool        GraphicsCommandPool;
VkPipeline           GraphicsPipeline;
VkPipelineCache      GraphicsPipelineCache;
VkPipelineLayout     GraphicsPipelineLayout;
uint32_t             Height;
VkInstance           Instance;
VkExtensionProperties * InstanceExtensions;
VkLayerProperties *  InstanceLayers;
VkLogicalDevice      LogicalDevice;
GLFWwindow *        MainWindow;
VkPhysicalDevice     PhysicalDevice;
VkPhysicalDeviceProperties PhysicalDeviceProperties;
uint32_t             PhysicalDeviceCount;
VkPhysicalDeviceFeatures PhysicalDeviceFeatures;
VkImage *            PresentImages;
VkImageView *        PresentImageViews;           // the swap chain image views
VkQueue              Queue;
VkRect2D             RenderArea;
VkRenderPass         RenderPass;
VkSemaphore          SemaphoreImageAvailable;
VkSemaphore          SemaphoreRenderFinished;
VkShaderModule       ShaderModuleFragment;
VkShaderModule       ShaderModuleVertex;
VkBuffer             StagingBuffer;
VkDeviceMemory       StagingBufferMemory;
VkSurfaceKHR         Surface;
VkSwapchainKHR       SwapChain;
VkCommandBuffer      TextureCommandBuffer;       // used for transferring texture from sta
ging buffer to actual texture buffer
VkImage              TextureImage;
VkDeviceMemory       TextureImageMemory;
VkCommandPool        TransferCommandPool;
VkDebugReportCallbackEXT WarningCallback;
uint32_t             Width;

#include "SampleVertexData.cpp"

// *****
// APPLICATION-RELATED GLOBAL VARIABLES:
// *****

int                ActiveButton;                 // current button that i
s down
FILE *             FpDebug;                       // where to send debugging messa
ges
struct lightBuf    Light;                        // cpu struct to hold light information
struct matBuf      Matrices;                     // cpu struct to hold matrix information
struct miscBuf     Misc;                         // cpu struct to hold miscellaneous info
rmation information
int                Mode;                          // 0 = use colors, 1 = u
se textures, ...

```

```
MyBuffer          MyLightUniformBuffer;
MyTexture         MyPuppyTexture;           // the cute puppy texture struct
MyBuffer          MyMatrixUniformBuffer;
MyBuffer          MyMiscUniformBuffer;
MyBuffer          MyVertexDataBuffer;
MyBuffer          MyJustIndexDataBuffer;
MyBuffer          MyJustVertexDataBuffer;
bool              NeedToExit;              // true means the program should
    exit
int
int              NumInstances;
int              NumRenders;              // how many times the re
nder loop has been called
bool            Paused;                   // true means don't animate
float           Scale;                     // scaling factor
double          Time;
bool            Verbose;                   // true = write messages into a
file
int             Xmouse, Ymouse;           // mouse values
float           Xrot, Yrot;                // rotation angles in degrees
bool           UseIndexBuffer;            // true = use both vertex and in
dex buffer, false = just use vertex buffer
bool           UseLighting;               // true = use lighting for displ
ay
bool           UseRotate;                 // true = rotate-animate, false
= use mouse for interaction
```

```
// *****  
// FUNCTION PROTOTYPES:  
// *****  
  
VkResult DestroyAllVulkan( );  
  
//VkBool32 ErrorCallback( VkDebugReportFlagsEXT, VkDebugReportObjectTypeEXT  
, uint64_t, size_t, int32_t, const char *, const char *, void * );  
  
int FindMemoryThatIsDeviceLocal( uint32_t );  
int FindMemoryThatIsHostVisible( uint32_t );  
int FindMemoryByFlagAndType( VkMemoryPropertyFlags, uint32_t );  
  
int FindQueueFamilyThatDoesGraphics( );  
int FindQueueFamilyThatDoesCompute( );  
int FindQueueFamilyThatDoesTransfer( );  
  
void InitGraphics( );  
  
VkResult Init01Instance( );  
  
VkResult Init02CreateDebugCallbacks( );  
  
VkResult Init03PhysicalDeviceAndGetQueueFamilyProperties( );  
  
VkResult Init04LogicalDeviceAndQueue( );  
  
VkResult Init05DataBuffer( VkDeviceSize, VkBufferUsageFlags, OUT MyBuffer * );  
VkResult Init05UniformBuffer( VkDeviceSize, OUT MyBuffer * );  
VkResult Init05MyIndexDataBuffer( VkDeviceSize, OUT MyBuffer * );  
VkResult Init05MyVertexDataBuffer( VkDeviceSize, OUT MyBuffer * );  
VkResult Fill05DataBuffer( IN MyBuffer, IN void * );  
  
VkResult Init06CommandPools( );  
VkResult Init06CommandBuffers( );  
  
VkResult Init07TextureSampler( OUT MyTexture * );  
VkResult Init07TextureBuffer( INOUT MyTexture * );  
  
VkResult Init07TextureBufferAndFillFromBmpFile( IN std::string, OUT MyTexture * );  
  
VkResult Init08Swapchain( );  
  
VkResult Init09DepthStencilImage( );  
  
VkResult Init10RenderPasses( );  
  
VkResult Init11Framebuffers( );  
  
VkResult Init12SpirvShader( std::string, OUT VkShaderModule * );  
  
VkResult Init13DescriptorSetPool( );  
VkResult Init13DescriptorSetLayouts( );  
VkResult Init13DescriptorSets( );  
  
VkResult Init14GraphicsPipelineLayout( );  
VkResult Init14GraphicsVertexFragmentPipeline( VkShaderModule, VkShaderModule, VkPrimitiveTopology, OUT VkPipeline * );  
VkResult Init14ComputePipeline( VkShaderModule, OUT VkPipeline * );  
  
VkResult RenderScene( );  
void UpdateScene( );  
//VkBool32 WarningCallback( VkDebugReportFlagsEXT, VkDebugReportObjectTypeEXT, uint64_t, size_t, int32_t, const char *, const char *, void * );  
  
void PrintVkError( VkResult, std::string = "" );  
void Reset( );  
  
void InitGLFW( );
```



```
void          InitGLFWSurface( );
void          GLFWErrorCallback( int, const char * );
void          GLFWKeyboard( GLFWwindow *, int, int, int, int );
void          GLFWMouseButton( GLFWwindow *, int, int, int );
void          GLFWMouseMotion( GLFWwindow *, double, double );
double       GLFWGetTime( );

int          ReadInt( FILE * );
short       ReadShort( FILE * );
```

```
// *****
// MAIN PROGRAM:
// *****

int
main( int argc, char * argv[ ] )
{
    Width = 1536;
    Height = 1536;;

#ifdef _WIN32
    errno_t err = fopen_s( &FpDebug, DEBUGFILE, "w" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open debug print file '%s'\n", DEBUGFILE );
        FpDebug = stderr;
    }
#else
    FpDebug = fopen( DEBUGFILE, "w" );
    if( FpDebug == NULL )
    {
        fprintf( stderr, "Cannot open debug print file '%s'\n", DEBUGFILE );
        FpDebug = stderr;
    }
#endif
    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( );
        if( NeedToExit )
            break;
    }

    fprintf(FpDebug, "Closing the GLFW window\n");

    vkQueueWaitIdle( Queue );
    vkDeviceWaitIdle( LogicalDevice );
    DestroyAllVulkan( );
    glfwDestroyWindow( MainWindow );
    glfwTerminate( );
    return 0;
}
```

```
void
InitGraphics( )
{
    HERE_I_AM( "InitGraphics" );

    VkResult result = VK_SUCCESS;

    InitGLFW( );

    Init01Instance( );

    InitGLFWSurface( );

    Init02CreateDebugCallbacks( );

    Init03PhysicalDeviceAndGetQueueFamilyProperties( );

    Init04LogicalDeviceAndQueue( );

    Init05UniformBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );

    Init05UniformBuffer( sizeof(Light), &MyLightUniformBuffer );
    Fill05DataBuffer( MyLightUniformBuffer, (void *) &Light );

    Init05UniformBuffer( sizeof(Misc), &MyMiscUniformBuffer );
    Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );

    Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
    Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );

    Init05MyVertexDataBuffer( sizeof(JustVertexData), &MyJustVertexDataBuffer );
    Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );

    Init05MyIndexDataBuffer( sizeof(JustIndexData), &MyJustIndexDataBuffer );
    Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );

    Init06CommandPools();
    Init06CommandBuffers();

    Init07TextureSampler( &MyPuppyTexture );
    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_PRIMI
TIVE_TOPOLOGY_TRIANGLE_LIST, &GraphicsPipeline );
}
```

```

// *****
// CREATING THE INSTANCE:
// *****

VkResult
Init01Instance( )
{
    HERE_I_AM( "Init01Instance" );

    VkResult result = VK_SUCCESS;

    VkApplicationInfo vai;
    vai.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    vai.pNext = nullptr;
    vai.pApplicationName = "Vulkan Sample";
    vai.applicationVersion = 100;
    vai.pEngineName = "";
    vai.engineVersion = 1;
    vai.apiVersion = VK_MAKE_VERSION(1, 0, 0);

    // figure out what instance layers are wanted and available:

    std::vector<char *> instanceLayersWantedAndAvailable;

    {
        // these are the instance layers we would like to have:

        const char * instanceLayersWanted[ ] =
        {
            // "VK_LAYER_LUNARG_api_dump", // turn this on if want
to see each function call and its arguments (very slow!)
            "VK_LAYER_LUNARG_core_validation",
            "VK_LAYER_LUNARG_object_tracker",
            "VK_LAYER_LUNARG_parameter_validation",
            "VK_LAYER_NV_optimus"
        };
        uint32_t numLayersWanted = sizeof(instanceLayersWanted) / sizeof(char *);

        fprintf(FpDebug, "\n%d Instance Layers originally wanted:\n", numLayersWanted);
        for (unsigned int i = 0; i < numLayersWanted; i++)
        {
            fprintf(FpDebug, "\t%s\n", instanceLayersWanted[i]);
        }

        // see what layers are actually available:

        uint32_t numLayersAvailable;
        vkEnumerateInstanceLayerProperties( &numLayersAvailable, (VkLayerProperties *)nu
llptr );

        InstanceLayers = new VkLayerProperties[ numLayersAvailable ];
        result = vkEnumerateInstanceLayerProperties( &numLayersAvailable, InstanceLayers
);

        REPORT( "vkEnumerateInstanceLayerProperties" );
        if( result != VK_SUCCESS )
        {
            return result;
        }

        fprintf( FpDebug, "\n%d Instance Layers actually available:\n", numLayersAvailab
le );

        for( unsigned int i = 0; i < numLayersAvailable; i++ )
        {
            fprintf( FpDebug, "0x%08x %2d '%s' '%s'\n",
                    InstanceLayers[i].specVersion,
                    InstanceLayers[i].implementationVersion,
                    InstanceLayers[i].layerName,
                    InstanceLayers[i].description );
        }

        instanceLayersWantedAndAvailable.clear( );
        for( uint32_t wanted = 0; wanted < numLayersWanted; wanted++ )
        {

```

```

        for( uint32_t available = 0; available < numLayersAvailable; available++
)
    {
        if( strcmp( instanceLayersWanted[wanted], InstanceLayers[availab
le].layerName ) == 0 )
    {
        instanceLayersWantedAndAvailable.push_back( InstanceLaye
rs[available].layerName );
        break;
    }
    }

    fprintf( FpDebug, "\nWill now ask for %d Instance Layers:\n", (int)instanceLayer
sWantedAndAvailable.size( ) );
    for( uint32_t i = 0; i < instanceLayersWantedAndAvailable.size( ); i++ )
    {
        fprintf( FpDebug, "\t%s\n", instanceLayersWantedAndAvailable[i] );
    }
}

std::vector<char *> extensionsWantedAndAvailable;
{
    // figure out what instance extensions are wanted and available:
    const char * instanceExtensionsWanted[ ] =
    {
        "VK_KHR_surface",
#ifdef _WIN32
        "VK_KHR_win32_surface",
#endif
        "VK_EXT_debug_report",
    };
    uint32_t numExtensionsWanted = sizeof(instanceExtensionsWanted) / sizeof(char *)
;

    fprintf(FpDebug, "\n%d Instance Extensions originally wanted:\n", numExtensionsW
anted);
    for (unsigned int i = 0; i < numExtensionsWanted; i++)
    {
        fprintf(FpDebug, "\t%s\n", instanceExtensionsWanted[i]);
    }

    // see what extensions are actually available:
    uint32_t numExtensionsAvailable;
    vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensionsAvailable
, (VkExtensionProperties *)nullptr );
    InstanceExtensions = new VkExtensionProperties[ numExtensionsAvailable ];
    result = vkEnumerateInstanceExtensionProperties( (char *)nullptr, &numExtensions
Available, InstanceExtensions );
    REPORT( "vkEnumerateInstanceExtensionProperties" );
    if( result != VK_SUCCESS )
    {
        return result;
    }

    fprintf(FpDebug, "\n%d Instance Extensions actually available:\n", numExtensions
Available);
    for (unsigned int i = 0; i < numExtensionsAvailable; i++)
    {
        fprintf(FpDebug, "0x%08x '%s'\n",
            InstanceExtensions[i].specVersion,
            InstanceExtensions[i].extensionName);
    }

    // look for extensions both on the wanted list and the available list:

    extensionsWantedAndAvailable.clear( );
    for( uint32_t wanted = 0; wanted < numExtensionsWanted; wanted++ )
    {

```

```

        for( uint32_t available = 0; available < numExtensionsAvailable; availab
le++ )
        {
            if( strcmp( instanceExtensionsWanted[wanted], InstanceExtensions
[available].extensionName ) == 0 )
            {
                extensionsWantedAndAvailable.push_back( InstanceExtensio
ns[available].extensionName );
                break;
            }
        }

        fprintf( FpDebug, "\nWill now ask for %d Instance Extensions\n", (int) extension
sWantedAndAvailable.size( ) );
        for( uint32_t i = 0; i < extensionsWantedAndAvailable.size( ); i++ )
        {
            fprintf( FpDebug, "\t%s\n", extensionsWantedAndAvailable[i] );
        }

        // create the instance, asking for the layers and extensions:

        VkInstanceCreateInfo vici;
        vici.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
        vici.pNext = nullptr;
        vici.flags = 0;
        vici.pApplicationInfo = &vai;
        vici.enabledLayerCount      = (uint32_t) instanceLayersWantedAndAvailable.size(
);
        vici.ppEnabledLayerNames    = instanceLayersWantedAndAvailable.data( );
        vici.enabledExtensionCount  = (uint32_t) extensionsWantedAndAvailable.size( );
        vici.ppEnabledExtensionNames = extensionsWantedAndAvailable.data( );

        result = vkCreateInstance( IN &vici, PALLOCATOR, OUT &Instance );
        REPORT( "vkCreateInstance" );
        return result;
    }

    // *****
    // CREATE THE DEBUG CALLBACKS:
    // *****

    VkResult
    Init02CreateDebugCallbacks( )
    {
        HERE_I_AM( "Init02CreateDebugCallbacks" );

        VkResult result = VK_SUCCESS;

#ifdef NOTDEF
        PFN_vkCreateDebugReportCallbackEXT vkCreateDebugReportCallbackEXT = (PFN_vkCreateDebugRe
portCallbackEXT)nullptr;
        *(void **) &vkCreateDebugReportCallbackEXT = vkGetInstanceProcAddr( Instance, "vkCreateD
ebugReportCallbackEXT" );

        VkDebugReportCallbackCreateInfoEXT vdrcci;
        vdrcci.sType = VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT;
        vdrcci.pNext = nullptr;
        vdrcci.flags = VK_DEBUG_REPORT_ERROR_BIT_EXT;
        vdrcci.pfnCallback = (PFN_vkDebugReportCallbackEXT) &DebugReportCallback;
        vdrcci.pUserData = nullptr;

        result = vkCreateDebugReportCallbackEXT( Instance, IN &vdrcci, PALLOCATOR, OUT &ErrorCal
lback );
        REPORT( "vkCreateDebugReportCallbackEXT - 1" );

        vdrcci.flags = VK_DEBUG_REPORT_WARNING_BIT_EXT | VK_DEBUG_REPORT_PERFORMANCE_WAR
NING_BIT_EXT;

```

```
    result = vkCreateDebugReportCallbackEXT( Instance, IN &vdrcci, PALLOCATOR, OUT &WarningC
allback );
    REPORT( "vkCreateDebugReportCallbackEXT - 2" );
#endif

    return result;
}

#ifdef NOTYET
PFN_vkDebugReportCallbackEXT
DebugReportCallback(VkDebugReportFlagsEXT flags, VkDebugReportObjectTypeEXT objectType,
    uint64_t object, size_t location, int32_t messageCode,
    const char * pLayerPrefix, const char * pMessage, void * pUserData)
{
}

VkBool32
ErrorCallback( VkDebugReportFlagsEXT flags, VkDebugReportObjectTypeEXT objectType,
    uint64_t object, size_t location, int32_t messageCode,
    const char * pLayerPrefix, const char * pMessage, void * pUserData )
{
    fprintf( FpDebug, "ErrorCallback: ObjectType = 0x%0x ; object = %ld ; LayerPrefix = '%s'
; Message = '%s'\n", objectType, object, pLayerPrefix, pMessage );
    return VK_TRUE;
}

VkBool32
WarningCallback( VkDebugReportFlagsEXT flags, VkDebugReportObjectTypeEXT objectType,
    uint64_t object, size_t location, int32_t messageCode,
    const char * pLayerPrefix, const char * pMessage, void * pUserData )
{
    fprintf( FpDebug, "WarningCallback: ObjectType = 0x%0x ; object = %ld ; LayerPrefix = '%
s' ; Message = '%s'\n", objectType, object, pLayerPrefix, pMessage );
    return VK_TRUE;
}
#endif
```

```

// *****
// FINDING THE PHYSICAL DEVICES AND GET QUEUE FAMILY PROPERTIES:
// *****

VkResult
Init03PhysicalDeviceAndGetQueueFamilyProperties( )
{
    HERE_I_AM( "Init03PhysicalDeviceAndGetQueueFamilyProperties" );

    VkResult result = VK_SUCCESS;

    result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, (VkPhysicalDevic
ce *)nullptr );
    REPORT( "vkEnumeratePhysicalDevices - 1" );
    if( result != VK_SUCCESS || PhysicalDeviceCount <= 0 )
    {
        fprintf( FpDebug, "Could not count the physical devices\n" );
        return VK_SHOULD_EXIT;
    }

    fprintf( FpDebug, "\n%d physical devices found.\n", PhysicalDeviceCount);

    VkPhysicalDevice * physicalDevices = new VkPhysicalDevice[ PhysicalDeviceCount ];
    result = vkEnumeratePhysicalDevices( Instance, OUT &PhysicalDeviceCount, OUT physicalDev
ices );
    REPORT( "vkEnumeratePhysicalDevices - 2" );
    if( result != VK_SUCCESS )
    {
        fprintf( FpDebug, "Could not enumerate the %d physical devices\n", PhysicalDevic
eCount );
        return VK_SHOULD_EXIT;
    }

    int discreteSelect = -1;
    int integratedSelect = -1;
    for( unsigned int i = 0; i < PhysicalDeviceCount; i++ )
    {
        VkPhysicalDeviceProperties vpdp;
        vkGetPhysicalDeviceProperties( IN physicalDevices[i], OUT &vpdp );
        if( result != VK_SUCCESS )
        {
            fprintf( FpDebug, "Could not get the physical device properties of devic
e %d\n", i );
            return VK_SHOULD_EXIT;
        }

        fprintf( FpDebug, " \n\nDevice %2d:\n", i );
        fprintf( FpDebug, "\tAPI version: %d\n", vpdp.apiVersion );
        fprintf( FpDebug, "\tDriver version: %d\n", vpdp.apiVersion );
        fprintf( FpDebug, "\tVendor ID: 0x%04x\n", vpdp.vendorID );
        fprintf( FpDebug, "\tDevice ID: 0x%04x\n", vpdp.deviceID );
        fprintf( FpDebug, "\tPhysical Device Type: %d =", vpdp.deviceType );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU ) fprintf( FpDebug
, " (Discrete GPU)\n" );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU ) fprintf( FpDebug
, " (Integrated GPU)\n" );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU ) fprintf( FpDebug
, " (Virtual GPU)\n" );
        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU ) fprintf( FpDebug
, " (CPU)\n" );
        fprintf( FpDebug, "\tDevice Name: %s\n", vpdp.deviceName );
        fprintf( FpDebug, "\tPipeline Cache Size: %d\n", vpdp.pipelineCacheUUID[0] );
        //fprintf( FpDebug, "?", vpdp.limits );
        //fprintf( FpDebug, "?", vpdp.sparseProperties );

        // need some logical here to decide which physical device to select:

        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
            discreteSelect = i;

        if( vpdp.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
            integratedSelect = i;
    }
}

```



```

int which = -1;
if( discreteSelect >= 0 )
{
    which = discreteSelect;
    PhysicalDevice = physicalDevices[which];
}
else if( integratedSelect >= 0 )
{
    which = integratedSelect;
    PhysicalDevice = physicalDevices[which];
}
else
{
    fprintf( FpDebug, "Could not select a Physical Device\n" );
    return VK_SHOULD_EXIT;
}

delete[ ] physicalDevices;

vkGetPhysicalDeviceProperties( PhysicalDevice, OUT &PhysicalDeviceProperties );
fprintf( FpDebug, "Device #%d selected ('%s')\n", which, PhysicalDeviceProperties.device
Name );

vkGetPhysicalDeviceFeatures( IN PhysicalDevice, OUT &PhysicalDeviceFeatures );

fprintf( FpDebug, "\nPhysical Device Features:\n");
fprintf( FpDebug, "geometryShader = %2d\n", PhysicalDeviceFeatures.geometryShader);
fprintf( FpDebug, "tessellationShader = %2d\n", PhysicalDeviceFeatures.tessellationShade
r );
fprintf( FpDebug, "multiDrawIndirect = %2d\n", PhysicalDeviceFeatures.multiDrawIndirect
);
fprintf( FpDebug, "wideLines = %2d\n", PhysicalDeviceFeatures.wideLines );
fprintf( FpDebug, "largePoints = %2d\n", PhysicalDeviceFeatures.largePoints );
fprintf( FpDebug, "multiViewport = %2d\n", PhysicalDeviceFeatures.multiViewport );
fprintf( FpDebug, "occlusionQueryPrecise = %2d\n", PhysicalDeviceFeatures.occlusionQuery
Precise );
fprintf( FpDebug, "pipelineStatisticsQuery = %2d\n", PhysicalDeviceFeatures.pipelineStat
isticsQuery );
fprintf( FpDebug, "shaderFloat64 = %2d\n", PhysicalDeviceFeatures.shaderFloat64 );
fprintf( FpDebug, "shaderInt64 = %2d\n", PhysicalDeviceFeatures.shaderInt64 );
fprintf( FpDebug, "shaderInt16 = %2d\n", PhysicalDeviceFeatures.shaderInt16 );

#ifdef COMMENT
    All of these VkPhysicalDeviceFeatures are VkBool32s:
robustBufferAccess;
fullDrawIndexUint32;
imageCubeArray;
independentBlend;
geometryShader;
tessellationShader;
sampleRateShading;
dualSrcBlend;
logicOp;
multiDrawIndirect;
drawIndirectFirstInstance;
depthClamp;
depthBiasClamp;
fillModeNonSolid;
depthBounds;
wideLines;
largePoints;
alphaToOne;
multiViewport;
samplerAnisotropy;
textureCompressionETC2;
textureCompressionASTC_LDR;
textureCompressionBC;
occlusionQueryPrecise;
pipelineStatisticsQuery;
vertexPipelineStoresAndAtomics;
fragmentStoresAndAtomics;
shaderTessellationAndGeometryPointSize;
shaderImageGatherExtended;

```

```

shaderStorageImageExtendedFormats;
shaderStorageImageMultisample;
shaderStorageImageReadWithoutFormat;
shaderStorageImageWriteWithoutFormat;
shaderUniformBufferArrayDynamicIndexing;
shaderSampledImageArrayDynamicIndexing;
shaderStorageBufferArrayDynamicIndexing;
shaderStorageImageArrayDynamicIndexing;
shaderClipDistance;
shaderCullDistance;
shaderFloat64;
shaderInt64;
shaderInt16;
shaderResourceResidency;
shaderResourceMinLod;
sparseBinding;
sparseResidencyBuffer;
sparseResidencyImage2D;
sparseResidencyImage3D;
sparseResidency2Samples;
sparseResidency4Samples;
sparseResidency8Samples;
sparseResidency16Samples;
sparseResidencyAliased;
variableMultisampleRate;
inheritedQueries;
#endif

    VkFormatProperties                                     vfp;
#ifdef CHOICES
VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT = 0x00000001,
VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT = 0x00000002,
VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT = 0x00000004,
VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000008,
VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT = 0x00000010,
VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT = 0x00000020,
VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT = 0x00000040,
VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT = 0x00000080,
VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT = 0x00000100,
VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000200,
VK_FORMAT_FEATURE_BLIT_SRC_BIT = 0x00000400,
VK_FORMAT_FEATURE_BLIT_DST_BIT = 0x00000800,
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT = 0x00001000,
VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_CUBIC_BIT_IMG = 0x00002000,
#endif

    fprintf( FpDebug, "\nImage Formats Checked:\n" );
    vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R32G32B32A32_SFLOAT, &
vfp );
    fprintf( FpDebug, "Format VK_FORMAT_R32G32B32A32_SFLOAT: 0x%08x 0x%08x 0x%08x\n",
vfp.linearTilingFeatures, vfp.optimalTilingFeatures, vfp.bufferF
eatures );
    vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_R8G8B8A8_UNORM, &vfp );
;
    fprintf( FpDebug, "Format VK_FORMAT_R8G8B8A8_UNORM: 0x%08x 0x%08x 0x%08x\n",
vfp.linearTilingFeatures, vfp.optimalTilingFeatures, vfp.bufferF
eatures );
    vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_B8G8R8A8_UNORM, &vfp );
;
    fprintf( FpDebug, "Format VK_FORMAT_B8G8R8A8_UNORM: 0x%08x 0x%08x 0x%08x\n",
vfp.linearTilingFeatures, vfp.optimalTilingFeatures, vfp.bufferF
eatures );
    vkGetPhysicalDeviceFormatProperties( PhysicalDevice, IN VK_FORMAT_B8G8R8A8_SRGB, &vfp );
    fprintf( FpDebug, "Format VK_FORMAT_B8G8R8A8_SRGB: 0x%08x 0x%08x 0x%08x\n",
vfp.linearTilingFeatures, vfp.optimalTilingFeatures, vfp.bufferF
eatures );

    VkPhysicalDeviceMemoryProperties vpdmp;
    vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );

    fprintf( FpDebug, "\n%d Memory Types:\n", vpdmp.memoryTypeCount );
    for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
    {
        VkMemoryType vmt = vpdmp.memoryTypes[i];

```

```

        VkMemoryPropertyFlags vmpf = vmt.propertyFlags;
        fprintf( FpDebug, "Memory %2d: ", i );
        if( (vmpf & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT) != 0 ) fprintf( FpDebug
, " DeviceLocal" );
        if( (vmpf & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT) != 0 ) fprintf( FpDebug
, " HostVisible" );
        if( (vmpf & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT) != 0 ) fprintf( FpDebug
, " HostCoherent" );
        if( (vmpf & VK_MEMORY_PROPERTY_HOST_CACHED_BIT) != 0 ) fprintf( FpDebug
, " HostCached" );
        if( (vmpf & VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT) != 0 ) fprintf( FpDebug
, " LazilyAllocated" );
        fprintf(FpDebug, "\n");
    }

    fprintf( FpDebug, "\n%d Memory Heaps:\n", vpdmp.memoryHeapCount );
    for( unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ )
    {
        fprintf(FpDebug, "Heap %d: ", i);
        VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
        fprintf( FpDebug, " size = 0x%08lx", (unsigned long int)vmh.size );
        if( ( vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT ) != 0 ) fprintf(
FpDebug, " DeviceLocal" );
        if( ( vmh.flags & VK_MEMORY_HEAP_MULTI_INSTANCE_BIT ) != 0 ) fprintf( FpDebug
, " MultiInstance" );
        fprintf(FpDebug, "\n");
    }

    uint32_t count = -1;
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyP
roperties *)nullptr );
    fprintf( FpDebug, "\nFound %d Queue Families:\n", count );

    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );
    for( unsigned int i = 0; i < count; i++ )
    {
        fprintf( FpDebug, "\t%d: Queue Family Count = %2d ; ", i, vqfp[i].queueCount
);
        if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 ) fprintf( FpDebug
, " Graphics" );
        if( ( vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT ) != 0 ) fprintf( FpDebug
, " Compute" );
        if( ( vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 ) fprintf( FpDebug
, " Transfer" );
        fprintf(FpDebug, "\n");
    }

    delete[ ] vqfp;

    return result;
}

```

```

// *****
// CREATE THE LOGICAL DEVICE AND QUEUE:
// *****

VkResult
Init04LogicalDeviceAndQueue( )
{
    HERE_I_AM( "Init04LogicalDeviceAndQueue" );

    VkResult result = VK_SUCCESS;

    float    queuePriorities[ NUM_QUEUES_WANTED ] =
    {
        1.
    };

    VkDeviceQueueCreateInfo          vdqci[ NUM_QUEUES_WANTED ];
    vdqci[0].sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    vdqci[0].pNext = nullptr;
    vdqci[0].flags = 0;
    vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
    vdqci[0].queueCount = 1;           // how many queues to create
    vdqci[0].pQueuePriorities = queuePriorities;    // array of queue priorities [0.
,1.]

    const char * myDeviceLayers[ ] =
    {
        "VK_LAYER_LUNARG_object_tracker",
        "VK_LAYER_LUNARG_parameter_validation",
    };

    const char * myDeviceExtensions[ ] =
    {
        "VK_KHR_swapchain",
    };

    // see what device layers are available:

    uint32_t layerCount;
    vkEnumerateDeviceLayerProperties( PhysicalDevice, &layerCount, (VkLayerProperties *)nullptr);

    VkLayerProperties * deviceLayers = new VkLayerProperties[layerCount];
    result = vkEnumerateDeviceLayerProperties( PhysicalDevice, &layerCount, deviceLayers);
    REPORT("vkEnumerateDeviceLayerProperties");
    if (result != VK_SUCCESS)
    {
        return result;
    }

    fprintf(FpDebug, "\n%d physical device layers enumerated:\n", layerCount);
    for (unsigned int i = 0; i < layerCount; i++)
    {
        fprintf(FpDebug, "0x%08x  %2d  '%s'  '%s'\n",
            deviceLayers[i].specVersion,
            deviceLayers[i].implementationVersion,
            deviceLayers[i].layerName,
            deviceLayers[i].description);

        // see what device extensions are available:

        uint32_t extensionCount;
        vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].layerName,
        &extensionCount, (VkExtensionProperties *)nullptr);
        VkExtensionProperties * deviceExtensions = new VkExtensionProperties[extensionCo
unt];

        result = vkEnumerateDeviceExtensionProperties(PhysicalDevice, deviceLayers[i].la
yerName, &extensionCount, deviceExtensions);
        REPORT("vkEnumerateDeviceExtensionProperties");
        if (result != VK_SUCCESS)
        {

```

```
        return result;
    }

    fprintf(FpDebug, "\t%d device extensions enumerated for '%s':\n", extensionCount
, deviceLayers[i].layerName );
    for (unsigned int ii = 0; ii < extensionCount; ii++)
    {
        fprintf(FpDebug, "\t0x%08x  '%s'\n",
                deviceExtensions[ii].specVersion,
                deviceExtensions[ii].extensionName);
    }
    fprintf(FpDebug, "\n");
}

delete[ ] deviceLayers;

VkDeviceCreateInfo  vdci;
vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdci.pNext = nullptr;
vdci.flags = 0;
vdci.queueCreateInfoCount = NUM_QUEUES_WANTED;           // # of device queues, e
ach of which can create multiple queues
vdci.pQueueCreateInfos = IN &vdqci[0];                 // array of VkDeviceQueu
eCreateInfo's

vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
//vdci.enabledLayerCount = 0;
vdci.ppEnabledLayerNames = myDeviceLayers;

vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdci.ppEnabledExtensionNames = myDeviceExtensions;

vdci.pEnabledFeatures = IN &PhysicalDeviceFeatures;    // already created

result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice
);
REPORT( "vkCreateLogicalDevice" );

// get the queue for this logical device:
vkGetDeviceQueue( LogicalDevice, 0, 0, OUT &Queue );
// queueFamilyIndex, queueIndex
return result;
}
```

```

// *****
// CREATE A DATA BUFFER:
// *****

// This just creates the data buffer -- filling it with data uses the Fill05DataBuffer function
// Use this for vertex buffers, index buffers, uniform buffers, and textures

VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    HERE_I_AM( "Init05DataBuffer" );

    VkResult result = VK_SUCCESS;

    VkBufferCreateInfo vbci;
        vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
        vbci.pNext = nullptr;
        vbci.flags = 0;
        vbci.size = size;
        vbci.usage = usage;

#ifdef CHOICES
VK_USAGE_TRANSFER_SRC_BIT
VK_USAGE_TRANSFER_DST_BIT
VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
VK_USAGE_UNIFORM_BUFFER_BIT
VK_USAGE_STORAGE_BUFFER_BIT
VK_USAGE_INDEX_BUFFER_BIT
VK_USAGE_VERTEX_BUFFER_BIT
VK_USAGE_INDIRECT_BUFFER_BIT
#endif
        vbci.queueFamilyIndexCount = 0;
        vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
        vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE; // can only use CONCURRENT if .q
ueueFamilyIndexCount > 0
#ifdef CHOICES
VK_SHARING_MODE_EXCLUSIVE
VK_SHARING_MODE_CONCURRENT
#endif

        pMyBuffer->size = size;
        result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer )
;
        REPORT( "vkCreateBuffer" );

        VkMemoryRequirements vmr;
        vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );
// fills vmr
        if( Verbose )
        {
            fprintf( FpDebug, "Buffer vmr.size = %lld\n", vmr.size );
            fprintf( FpDebug, "Buffer vmr.alignment = %lld\n", vmr.alignment );
            fprintf( FpDebug, "Buffer vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits );
            fflush( FpDebug );
        }

        VkMemoryAllocateInfo vmai;
            vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
            vmai.pNext = nullptr;
            vmai.allocationSize = vmr.size;
            vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( vmr.memoryTypeBits );

        VkDeviceMemory vdm;
        result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
        REPORT( "vkAllocateMemory" );
        pMyBuffer->vdm = vdm;

        result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 );
// 0 is the offset
        REPORT( "vkBindBufferMemory" );

        return result;
}

```

```

// *****
// CREATE AN INDEX BUFFER:
// *****
// this allocates space for a data buffer, but doesn't yet fill it:
VkResult
Init05MyIndexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
    VkResult result = Init05DataBuffer(size, VK_BUFFER_USAGE_INDEX_BUFFER_BIT, pMyBuffer);
    // fills pMyBuffer
    REPORT("Init05MyIndexDataBufferBuffer");
    return result;
}

// *****
// CREATE A VERTEX BUFFER:
// *****
// this allocates space for a data buffer, but doesn't yet fill it:

VkResult
Init05MyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer )
;    // fills pMyBuffer
    REPORT( "InitDataBuffer" );
    return result;
}

// *****
// CREATE A UNIFORM BUFFER:
// *****
// this allocates space for a data buffer, but doesn't yet fill it:

VkResult
Init05UniformBuffer( VkDeviceSize size, MyBuffer * pMyBuffer )
{
    VkResult result = Init05DataBuffer( size, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, OUT pMyBuf
fer ); // fills pMyBuffer
    return result;
}

// *****
// FILL A DATA BUFFER:
// *****

VkResult
Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    // the size of the data had better match the size that was used to Init the buffer!

    void * pGpuMemory;
    vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, &pGpuMemory );
// 0 and 0 are offset and flags
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
    return VK_SUCCESS;

    // the way shown here makes it happen immediately
    //
    // but, we could use vkCmdUpdateBuffer( CommandBuffer, myBuffer.buffer, 0, myBuffer.size
, data );
    // instead, except that this requires use of the CommandBuffer
    // which might not be so bad since we are using the CommandBuffer at that moment to draw
    anyway
}

```

```

// *****
// CREATE A TEXTURE SAMPLER:
// *****

VkResult
Init07TextureSampler( MyTexture * pMyTexture )
{
    HERE_I_AM( "Init07TextureSampler" );

    VkResult result = VK_SUCCESS;

    VkSamplerCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    vsci.magFilter = VK_FILTER_LINEAR;
    vsci.minFilter = VK_FILTER_LINEAR;
    vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;

#ifdef CHOICES
VK_SAMPLER_ADDRESS_MODE_REPEAT
VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
#endif
        vsci.mipLodBias = 0.;
        vsci.anisotropyEnable = VK_FALSE;
        vsci.maxAnisotropy = 1.;
        vsci.compareEnable = VK_FALSE;
        vsci.compareOp = VK_COMPARE_OP_NEVER;

#ifdef CHOICES
VK_COMPARE_OP_NEVER
VK_COMPARE_OP_LESS
VK_COMPARE_OP_EQUAL
VK_COMPARE_OP_LESS_OR_EQUAL
VK_COMPARE_OP_GREATER
VK_COMPARE_OP_NOT_EQUAL
VK_COMPARE_OP_GREATER_OR_EQUAL
VK_COMPARE_OP_ALWAYS
#endif
        vsci.minLod = 0.;
        vsci.maxLod = 0.;
        vsci.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;

#ifdef CHOICES
VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
VK_BORDER_COLOR_INT_TRANSPARENT_BLACK
VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK
VK_BORDER_COLOR_INT_OPAQUE_BLACK
VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE
VK_BORDER_COLOR_INT_OPAQUE_WHITE
#endif
        vsci.unnormalizedCoordinates = VK_FALSE;           // VK_TRUE means we are use raw
    texels as the index                                   // VK_FALSE means we are using th
    e usual 0. - 1.

    result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampl
er );
    REPORT( "vkCreateSampler" );
    return result;
}

// *****
// CREATE A TEXTURE BUFFER:
// *****

// assume we get to here and have in a MyTexture struct:
// * an unsigned char array, holding the pixel rgba

```



```

//      * width  is the number of texels in s
//      * height is the number of texels in t

VkResult
Init07TextureBuffer( INOUT MyTexture * pMyTexture)
{
    HERE_I_AM( "Init07TextureBuffer" );

    VkResult result = VK_SUCCESS;

    uint32_t texWidth = pMyTexture->width;
    uint32_t texHeight = pMyTexture->height;
    unsigned char *texture = pMyTexture->pixels;
    VkDeviceSize textureSize = texWidth * texHeight * 4;           // rgba, 1 byte each

    VkImage stagingImage;
    VkImage textureImage;

    // *****
    // this first {...} is to create the staging image:
    // *****
    {
        VkImageCreateInfo vici;
        vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
        vici.pNext = nullptr;
        vici.flags = 0;

#ifdef CHOICES
        VK_IMAGE_CREATE_SPARSE_BINDING_BIT
        VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
        VK_IMAGE_CREATE_SPARSE_ALIASED_BIT
        VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT
        VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT
        VK_IMAGE_CREATE_BIND_SFR_BIT_KHR
        VK_IMAGE_CREATE_2D_ARRAY_COMPATIBLE_BIT_KHR
#endif

        vici.imageType = VK_IMAGE_TYPE_2D;
        //vici.format = VK_FORMAT_R8G8B8A8_UNORM;
        vici.format = VK_FORMAT_B8G8R8A8_SRGB;
        vici.extent.width = texWidth;
        vici.extent.height = texHeight;
        vici.extent.depth = 1;
        vici.mipLevels = 1;
        vici.arrayLayers = 1;
        vici.samples = VK_SAMPLE_COUNT_1_BIT;
        vici.tiling = VK_IMAGE_TILING_LINEAR;

#ifdef CHOICES
        VK_IMAGE_TILING_OPTIMAL
        VK_IMAGE_TILING_LINEAR
#endif

        vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;

#ifdef CHOICES
        VK_IMAGE_USAGE_TRANSFER_SRC_BIT
        VK_IMAGE_USAGE_TRANSFER_DST_BIT
        VK_IMAGE_USAGE_SAMPLED_BIT
        VK_IMAGE_USAGE_STORAGE_BIT
        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
        VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
        VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
        VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
#endif

        vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
        vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

#ifdef CHOICES
        VK_IMAGE_LAYOUT_UNDEFINED
        VK_IMAGE_LAYOUT_PREINITIALIZED
#endif

        vici.queueFamilyIndexCount = 0;
        vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

        result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage);
    // allocated, but not filled
        REPORT("vkCreateImage");
    }
}

```

```

VkMemoryRequirements                                vmr;
vkGetImageMemoryRequirements(LogicalDevice, IN stagingImage, OUT &vmr);

if (Verbose)
{
    fprintf(FpDebug, "Image vmr.size = %lld\n", vmr.size);
    fprintf(FpDebug, "Image vmr.alignment = %lld\n", vmr.alignment);
    fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
}

ts);

    fflush(FpDebug);
}

VkMemoryAllocateInfo                                vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( vmr.memoryTypeBits );
; // because we want to mmap it

VkDeviceMemory                                    vdm;
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
REPORT("vkAllocateMemory");
pMyTexture->vdm = vdm;

result = vkBindImageMemory(LogicalDevice, IN stagingImage, IN vdm, 0); // 0 = 0
ffset
REPORT("vkBindImageMemory");

// we have now created the staging image -- fill it with the pixel data:

VkImageSubresource                                vis;
vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
vis.mipLevel = 0;
vis.arrayLayer = 0;

VkSubresourceLayout                                vsl;
vkGetImageSubresourceLayout(LogicalDevice, stagingImage, IN &vis, OUT &vsl);

if (Verbose)
{
    fprintf(FpDebug, "Subresource Layout:\n");
    fprintf(FpDebug, "\toffset = %lld\n", vsl.offset);
    fprintf(FpDebug, "\tsize = %lld\n", vsl.size);
    fprintf(FpDebug, "\trowPitch = %lld\n", vsl.rowPitch);
    fprintf(FpDebug, "\tarrayPitch = %lld\n", vsl.arrayPitch);
    fprintf(FpDebug, "\tdepthPitch = %lld\n", vsl.depthPitch);
    fflush(FpDebug);
}

void * gpuMemory;
vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
// 0 and 0 = offset and memory map flags

if (vsl.rowPitch == 4 * texWidth)
{
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
}
else
{
    unsigned char *gpuBytes = (unsigned char *)gpuMemory;
    for (unsigned int y = 0; y < texHeight; y++)
    {
        memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth],
(size_t)(4*texWidth) );
    }
}

vkUnmapMemory(LogicalDevice, vdm);

}
// *****

```

```

// *****
// this second {...} is to create the actual texture image:
// *****
{
    VkImageCreateInfo          vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.format = VK_FORMAT_R8G8B8A8_SRGB;
    vici.extent.width = texWidth;
    vici.extent.height = texHeight;
    vici.extent.depth = 1;
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_OPTIMAL;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;

T;
    // because we are transferring into it and will eventual
sample from it
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage);
// allocated, but not filled
    REPORT("vkCreateImage");

    VkMemoryRequirements          vmr;
    vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmr);

    if( Verbose )
    {
        fprintf( FpDebug, "Texture vmr.size = %lld\n", vmr.size );
        fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment );
        fprintf( FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits );
eBits );
        fflush( FpDebug );
    }

    VkMemoryAllocateInfo          vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal( vmr.memoryTypeBits );
; // because we want to sample from it

    VkDeviceMemory          vdm;
    result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
    REPORT("vkAllocateMemory");

    result = vkBindImageMemory( LogicalDevice, IN textureImage, IN vdm, 0 );
// 0 = offset
    REPORT( "vkBindImageMemory" );
}
// *****

// copy pixels from the staging image to the texture:

VkCommandBufferBeginInfo          vcbbi;
vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
vcbbi.pNext = nullptr;
vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;

result = vkBeginCommandBuffer( TextureCommandBuffer, IN &vcbbi);
REPORT( "Init07TextureBuffer -- vkBeginCommandBuffer" );

// *****
// transition the staging buffer layout:

```



```

        vo3.z = 0;

    VkExtent3D          ve3;
        ve3.width = texWidth;
        ve3.height = texHeight;
        ve3.depth = 1;

    VkImageCopy         vic;
        vic.srcSubresource = visl;
        vic.srcOffset = vo3;
        vic.dstSubresource = visl;
        vic.dstOffset = vo3;
        vic.extent = ve3;

    vkCmdCopyImage(TextureCommandBuffer,
        stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
        textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
        1, IN &vic);
}
// *****

// *****
// transition the texture buffer layout a second time:
// *****
{
    VkImageSubresourceRange    visr;
        visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        visr.baseMipLevel = 0;
        visr.levelCount = 1;
        visr.baseArrayLayer = 0;
        visr.layerCount = 1;

    VkImageMemoryBarrier      vimb;
        vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
        vimb.pNext = nullptr;
        vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
        vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
        vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        vimb.image = textureImage;
        vimb.srcAccessMask = 0;
        vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
        vimb.subresourceRange = visr;

    vkCmdPipelineBarrier(TextureCommandBuffer,
        VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0

        0, (VkMemoryBarrier *)nullptr,
        0, (VkBufferMemoryBarrier *)nullptr,
        1, IN &vimb);
}
// *****

result = vkEndCommandBuffer( TextureCommandBuffer );
REPORT("Init07TextureBuffer -- vkEndCommandBuffer");

VkSubmitInfo          vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = &TextureCommandBuffer;
    vsi.waitSemaphoreCount = 0;
    vsi.pWaitSemaphores = (VkSemaphore *)nullptr;
    vsi.signalSemaphoreCount = 0;
    vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
    vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;

result = vkQueueSubmit( Queue, 1, IN &vsi, VK_NULL_HANDLE );
if (Verbose)          REPORT("vkQueueSubmit");

result = vkQueueWaitIdle( Queue );
if (Verbose)          REPORT("vkQueueWaitIdle");

```

```

    // create an image view for the texture image:

    VkImageSubresourceRange          visr;
        visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        visr.baseMipLevel = 0;
        visr.levelCount = 1;
        visr.baseArrayLayer = 0;
        visr.layerCount = 1;

    VkImageViewCreateInfo            vivci;
        vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        vivci.pNext = nullptr;
        vivci.flags = 0;
        vivci.image = textureImage;
        vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
        vivci.format = VK_FORMAT_R8G8B8A8_SRGB;
        vivci.components.r = VK_COMPONENT_SWIZZLE_R;
        vivci.components.g = VK_COMPONENT_SWIZZLE_G;
        vivci.components.b = VK_COMPONENT_SWIZZLE_B;
        vivci.components.a = VK_COMPONENT_SWIZZLE_A;
        vivci.subresourceRange = visr;

    result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
    REPORT("vkCreateImageView");

    vkDestroyImage( LogicalDevice, stagingImage, PALLOCATOR );

    return result;
}

// *****
// CREATE A TEXTURE IMAGE FROM A BMP FILE:
// *****

VkResult
Init07TextureBufferAndFillFromBmpFile( IN std::string filename, OUT MyTexture * pMyTexture )
{
    HERE_I_AM( "Init07TextureBufferAndFillFromBmpFile" );

    VkResult result = VK_SUCCESS;

    const int birgb = { 0 };

    FILE * fp;
#ifdef WIN32
    _errno_t err = fopen_s( &fp, filename.c_str(), "rb" );
    if( err != 0 )
    {
        fprintf( stderr, "Cannot open BMP file '%s'\n", filename.c_str() );
        return VK_FAILURE;
    }
#else
    fp = fopen( filename.c_str(), "rb" );
    if( fp == NULL )
    {
        fprintf( stderr, "Cannot open BMP file '%s'\n", filename.c_str() );
        return VK_FAILURE;
    }
#endif

    FileHeader.bfType = ReadShort( fp );

    // if bfType is not 0x4d42, the file is not a bmp:

    if( FileHeader.bfType != 0x4d42 )
    {
        fprintf( FpDebug, "Wrong type of file: 0x%0x\n", FileHeader.bfType );
        fclose( fp );
        return VK_FAILURE;
    }
}

```

```

FileHeader.bfSize = ReadInt( fp );
FileHeader.bfReserved1 = ReadShort( fp );
FileHeader.bfReserved2 = ReadShort( fp );
FileHeader.bfOffBits = ReadInt( fp );

InfoHeader.biSize = ReadInt( fp );
InfoHeader.biWidth = ReadInt( fp );
InfoHeader.biHeight = ReadInt( fp );

uint32_t texWidth = InfoHeader.biWidth;
uint32_t texHeight = InfoHeader.biHeight;

InfoHeader.biPlanes = ReadShort( fp );
InfoHeader.biBitCount = ReadShort( fp );
InfoHeader.biCompression = ReadInt( fp );
InfoHeader.biSizeImage = ReadInt( fp );
InfoHeader.biXPelsPerMeter = ReadInt( fp );
InfoHeader.biYPelsPerMeter = ReadInt( fp );
InfoHeader.biClrUsed = ReadInt( fp );
InfoHeader.biClrImportant = ReadInt( fp );

fprintf( FpDebug, "Image size found: %d x %d\n", texWidth, texHeight );

unsigned char * texture = new unsigned char[ 4 * texWidth * texHeight ];

// extra padding bytes:
int numExtra = 4*(( 3*InfoHeader.biWidth)+3)/4 - 3*InfoHeader.biWidth;

// we do not support compression:
if( InfoHeader.biCompression != birgb )
{
    fprintf( FpDebug, "Wrong type of image compression: %d\n", InfoHeader.biCompress
ion );
    fclose( fp );
    return VK_FAILURE;
}

rewind( fp );
fseek( fp, 14+40, SEEK_SET );

if( InfoHeader.biBitCount == 24 )
{
    unsigned char *tp = texture;
    for( unsigned int t = 0; t < texHeight; t++ )
    {
        for( unsigned int s = 0; s < texWidth; s++, tp += 4 )
        {
            *(tp+3) = 255;           // a
            *(tp+2) = fgetc( fp );  // b
            *(tp+1) = fgetc( fp );  // g
            *(tp+0) = fgetc( fp );  // r
        }

        for( int e = 0; e < numExtra; e++ )
        {
            fgetc( fp );
        }
    }
}
fclose( fp );

pMyTexture->width = texWidth;
pMyTexture->height = texHeight;
pMyTexture->pixels = texture;

result = Init07TextureBuffer( INOUT pMyTexture );
REPORT( "Init07TextureBuffer" );

return result;
}

```

```
int
ReadInt( FILE *fp )
{
    unsigned char b3, b2, b1, b0;
    b0 = fgetc( fp );
    b1 = fgetc( fp );
    b2 = fgetc( fp );
    b3 = fgetc( fp );
    return ( b3 << 24 ) | ( b2 << 16 ) | ( b1 << 8 ) | b0;
}

short
ReadShort( FILE *fp )
{
    unsigned char b1, b0;
    b0 = fgetc( fp );
    b1 = fgetc( fp );
    return ( b1 << 8 ) | b0;
}
```



```

// *****
// CREATING THE SWAP CHAIN:
// *****

VkResult
Init08Swapchain( )
{
    HERE_I_AM( "Init08Swapchain" );

    VkResult result = VK_SUCCESS;

    VkSurfaceCapabilitiesKHR          vsc;

    vkGetPhysicalDeviceSurfaceCapabilitiesKHR( PhysicalDevice, Surface, OUT &vsc );
#ifdef ELEMENTS
    vsc.uint32_t                      minImageCount;
    vsc.uint32_t                      maxImageCount;
    vsc.VkExtent2D                   currentExtent;
    vsc.VkExtent2D                   minImageExtent;
    vsc.VkExtent2D                   maxImageExtent;
    vsc.uint32_t                      maxImageArrayLayers;
    vsc.VkSurfaceTransformFlagsKHR    supportedTransforms;
    vsc.VkSurfaceTransformFlagBitsKHR currentTransform;
    vsc.VkCompositeAlphaFlagsKHR      supportedCompositeAlpha;
    vsc.VkImageUsageFlags             supportedUsageFlags;
#endif

    VkExtent2D surfaceRes = vsc.currentExtent;
    fprintf( FpDebug, "\nvkGetPhysicalDeviceSurfaceCapabilitiesKHR:\n" );
    fprintf( FpDebug, "\tminImageCount = %d ; maxImageCount = %d\n", vsc.minImageCount, vsc.
maxImageCount );
    fprintf( FpDebug, "\tcurrentExtent = %d x %d\n", vsc.currentExtent.width, vsc.currentExt
ent.height );
    fprintf( FpDebug, "\tminImageExtent = %d x %d\n", vsc.minImageExtent.width, vsc.minImage
Extent.height );
    fprintf( FpDebug, "\tmaxImageExtent = %d x %d\n", vsc.maxImageExtent.width, vsc.maxImage
Extent.height );
    fprintf( FpDebug, "\tmaxImageArrayLayers = %d\n", vsc.maxImageArrayLayers );
    fprintf( FpDebug, "\tsupportedTransforms = 0x%04x\n", vsc.supportedTransforms );
    fprintf( FpDebug, "\tcurrentTransform = 0x%04x\n", vsc.currentTransform );
    fprintf( FpDebug, "\tsupportedCompositeAlpha = 0x%04x\n", vsc.supportedCompositeAlpha );
    fprintf( FpDebug, "\tsupportedUsageFlags = 0x%04x\n", vsc.supportedUsageFlags );

    VkBool32 supported;
    result = vkGetPhysicalDeviceSurfaceSupportKHR( PhysicalDevice, FindQueueFamilyThatDoesGr
aphics( ), Surface, &supported );
    REPORT( "vkGetPhysicalDeviceSurfaceSupportKHR" );
    if( supported == VK_TRUE )
    {
        fprintf( FpDebug, "*** This Surface is supported by the Graphics Queue **\n" );
    }
    else
    {
        fprintf( FpDebug, "*** This Surface is not supported by the Graphics Queue **\n"
);
    }
}

uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, (VkSurfaceF
ormatKHR *) nullptr );
VkSurfaceFormatKHR * surfaceFormats = new VkSurfaceFormatKHR[ formatCount ];
vkGetPhysicalDeviceSurfaceFormatsKHR( PhysicalDevice, Surface, &formatCount, surfaceForm
ats );
#ifdef FORMAT_CHOICES
VK_FORMAT_R8G8B8A8_UNORM = 37,
VK_FORMAT_R8G8B8A8_SNORM = 38,
VK_FORMAT_R8G8B8A8_USCALED = 39,
VK_FORMAT_R8G8B8A8_SSCALED = 40,
VK_FORMAT_R8G8B8A8_UINT = 41,
VK_FORMAT_R8G8B8A8_SINT = 42,
VK_FORMAT_R8G8B8A8_SRGB = 43,
VK_FORMAT_B8G8R8A8_UNORM = 44, <----- the 1080tis can use this one

```

```

VK_FORMAT_B8G8R8A8_SNORM = 45,
VK_FORMAT_B8G8R8A8_USCALED = 46,
VK_FORMAT_B8G8R8A8_SSCALED = 47,
VK_FORMAT_B8G8R8A8_UINT = 48,
VK_FORMAT_B8G8R8A8_SINT = 49,
VK_FORMAT_B8G8R8A8_SRGB = 50, <----- the 1080 tis can use this one
#endif

#ifdef COLOR_SPACE_CHOICES
VK_COLOR_SPACE_SRGB_NONLINEAR_KHR = 0, <----- the 1080tis use this one
VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT = 1000104001,
VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT = 1000104002,
VK_COLOR_SPACE_DCI_P3_LINEAR_EXT = 1000104003,
VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT = 1000104004,
VK_COLOR_SPACE_BT709_LINEAR_EXT = 1000104005,
VK_COLOR_SPACE_BT709_NONLINEAR_EXT = 1000104006,
VK_COLOR_SPACE_BT2020_LINEAR_EXT = 1000104007,
VK_COLOR_SPACE_HDR10_ST2084_EXT = 1000104008,
VK_COLOR_SPACE_DOLBYVISION_EXT = 1000104009,
VK_COLOR_SPACE_HDR10_HLG_EXT = 1000104010,
#endif

    fprintf( FpDebug, "\nFound %d Surface Formats:\n", formatCount );
    for( uint32_t i = 0; i < formatCount; i++ )
    {
        fprintf( FpDebug, "%3d:      %4d      %12d", i, surfaceFormats[i].format, surfaceF
ormats[i].colorSpace );\
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_SRGB_NONLINEAR_KHR\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_DISPLAY_P3_NONLINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_EXTENDED_SRGB_LINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_DCI_P3_LINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_DCI_P3_LINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_DCI_P3_NONLINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_BT709_LINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_BT709_LINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_BT709_NONLINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_BT709_NONLINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_BT2020_LINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_BT2020_LINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_HDR10_ST2084_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_HDR10_ST2084_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_DOLBYVISION_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_DOLBYVISION_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_HDR10_HLG_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_HDR10_HLG_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_ADOBERGB_LINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_ADOBERGB_LINEAR_EXT\n" );
        if (surfaceFormats[i].colorSpace == VK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT)
            fprintf( FpDebug, "\tVK_COLOR_SPACE_ADOBERGB_NONLINEAR_EXT\n" );
    }
    delete [ ] surfaceFormats;

    uint32_t presentModeCount;
    vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, (
VkPresentModeKHR *) nullptr );
    VkPresentModeKHR * presentModes = new VkPresentModeKHR[ presentModeCount ];
    vkGetPhysicalDeviceSurfacePresentModesKHR( PhysicalDevice, Surface, &presentModeCount, p
resentModes );
#ifdef PRESENT_MODE_CHOICES
VK_PRESENT_MODE_IMMEDIATE_KHR = 0,
VK_PRESENT_MODE_MAILBOX_KHR = 1, <----- the 1080tis can handle this
VK_PRESENT_MODE_FIFO_KHR = 2, <----- the 1080tis can handle this
VK_PRESENT_MODE_FIFO_RELAXED_KHR = 3, <----- the 1080tis can handle this
VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR = 1000111000,
VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR = 1000111001,
#endif
    fprintf( FpDebug, "\nFound %d Present Modes:\n", presentModeCount );
    for( uint32_t i = 0; i < presentModeCount; i++ )
    {

```

```

        fprintf( FpDebug, "%3d:      %4d", i, presentModes[i] );
        if( presentModes[i] == VK_PRESENT_MODE_IMMEDIATE_KHR )           fprintf(
FpDebug, "\tVK_PRESENT_MODE_IMMEDIATE_KHR\n" );
        if( presentModes[i] == VK_PRESENT_MODE_MAILBOX_KHR )           fprintf(
FpDebug, "\tVK_PRESENT_MODE_MAILBOX_KHR\n" );
        if( presentModes[i] == VK_PRESENT_MODE_FIFO_KHR )              fprintf(
FpDebug, "\tVK_PRESENT_MODE_FIFO_KHR\n" );
        if( presentModes[i] == VK_PRESENT_MODE_FIFO_RELAXED_KHR )      fprintf(
FpDebug, "\tVK_PRESENT_MODE_FIFO_RELAXED_KHR\n" );
        if( presentModes[i] == VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR ) fprintf(
FpDebug, "\tVK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR\n" );
        if( presentModes[i] == VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR ) fprintf(
FpDebug, "\tVK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR\n" );
    }
    fprintf( stderr, "\n" );

    // find the present mode we should use:

    VkPresentModeKHR acceptablePresentModes[ ] =
    {
        VK_PRESENT_MODE_MAILBOX_KHR,
        VK_PRESENT_MODE_FIFO_KHR,
        VK_PRESENT_MODE_FIFO_RELAXED_KHR,
        VK_PRESENT_MODE_IMMEDIATE_KHR,
    };

    VkPresentModeKHR thePresentMode = (VkPresentModeKHR) VK_NULL_HANDLE;
    for( VkPresentModeKHR apm : acceptablePresentModes )
    {
        for( uint32_t i = 0; i < presentModeCount; i++ )
        {
            if( apm == presentModes[i] )
            {
                thePresentMode = apm;
                break;
            }
        }
        if (thePresentMode != (VkPresentModeKHR)VK_NULL_HANDLE)
            break;
    }

    if( thePresentMode == (VkPresentModeKHR) VK_NULL_HANDLE )
    {
        fprintf( FpDebug, "Couldn't find an acceptable Present Mode!\n" );
    }
    else
    {
        fprintf( FpDebug, "The Present Mode to use = %d\n", thePresentMode );
    }

    delete [ ] presentModes;

#ifdef ELEMENTS
    surfaceRes.width
    surfaceRes.height;
#endif

    VkSwapchainCreateInfoKHR          vscci;
    vscci.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
    vscci.pNext = nullptr;
    vscci.flags = 0;
    vscci.surface = Surface;
    vscci.minImageCount = 2;
    vscci.imageFormat = VK_FORMAT_B8G8R8A8_UNORM; // double buffering
    vscci.imageFormat = VK_FORMAT_B8G8R8A8_SRGB;
    vscci.imageColorSpace = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
    vscci.imageExtent.width = surfaceRes.width;
    vscci.imageExtent.height = surfaceRes.height;
    vscci.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
    vscci.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
    vscci.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;

```

```

        vscci.imageArrayLayers = 1;
        vscci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
        vscci.queueFamilyIndexCount = 0;
        vscci.pQueueFamilyIndices = (const uint32_t *)nullptr;
        //vscci.presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
        vscci.presentMode = thePresentMode;
        vscci.oldSwapchain = VK_NULL_HANDLE;    // we're not replacing an old swapchain
        vscci.clipped = VK_TRUE;

    result = vkCreateSwapchainKHR( LogicalDevice, IN &vscci, PALLOCATOR, OUT &SwapChain );
    REPORT( "vkCreateSwapchainKHR" );

    uint32_t imageCount;
    result = vkGetSwapchainImagesKHR( LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage
*)nullptr );    // 0
    REPORT( "vkGetSwapchainImagesKHR - 0" );
    if( imageCount != 2 )
    {
        fprintf( FpDebug, "imageCount return from vkGetSwapchainImages = %d; should have
been 2\n", imageCount );
        return result;
    }

    PresentImages = new VkImage[ imageCount ];
    result = vkGetSwapchainImagesKHR( LogicalDevice, SwapChain, OUT &imageCount, PresentImag
es );    // 0
    REPORT( "vkGetSwapchainImagesKHR - 1" );

    // present views for the double-buffering:

    PresentImageViews = new VkImageView[ imageCount ];    // better be 2
    for( unsigned int i = 0; i < imageCount; i++ )
    {
        VkImageViewCreateInfo    vivci;
        vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        vivci.pNext = nullptr;
        vivci.flags = 0;
        vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
        //vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
        vivci.format = VK_FORMAT_B8G8R8A8_SRGB;
        vivci.components.r = VK_COMPONENT_SWIZZLE_R;
        vivci.components.g = VK_COMPONENT_SWIZZLE_G;
        vivci.components.b = VK_COMPONENT_SWIZZLE_B;
        vivci.components.a = VK_COMPONENT_SWIZZLE_A;
        vivci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        vivci.subresourceRange.baseMipLevel = 0;
        vivci.subresourceRange.levelCount = 1;
        vivci.subresourceRange.baseArrayLayer = 0;
        vivci.subresourceRange.layerCount = 1;
        vivci.image = PresentImages[i];

        result = vkCreateImageView( LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentIm
ageViews[i] );
        REPORT( "vkCreateImageView" );
    }

    return result;
}

```

```

// *****
// CREATING THE DEPTH AND STENCIL IMAGE:
// *****

VkResult
Init09DepthStencilImage( )
{
    HERE_I_AM( "Init09DepthStencilImage" );

    VkResult result = VK_SUCCESS;

    VkExtent3D ve3d = { Width, Height, 1 };

    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.format = VK_FORMAT_D32_SFLOAT_S8_UINT;
    vici.extent = ve3d;
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_OPTIMAL;
    vici.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
    vici.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

    result = vkCreateImage( LogicalDevice, IN &vici, PALLOCATOR, &DepthStencilImage );
    REPORT( "vkCreateImage" );

    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements( LogicalDevice, IN DepthStencilImage, OUT &vmr );

    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal( vmr.memoryTypeBits );

    VkDeviceMemory imageMemory;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &imageMemory );
    REPORT( "vkAllocateMemory" );

    result = vkBindImageMemory( LogicalDevice, DepthStencilImage, imageMemory, 0 ); // 0 is
the offset
    REPORT( "vkBindImageMemory" );

    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.image = DepthStencilImage;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = vici.format;
    vivci.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
    vivci.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
    vivci.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
    vivci.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
    vivci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
    vivci.subresourceRange.baseMipLevel = 0;
    vivci.subresourceRange.levelCount = 1;
    vivci.subresourceRange.baseArrayLayer = 0;
    vivci.subresourceRange.layerCount = 1;

    result = vkCreateImageView( LogicalDevice, IN &vivci, PALLOCATOR, OUT &DepthSten
cilImageView );
    REPORT( "vkCreateImageView" );
    return result;
}

```



```

// *****
// CREATING THE RENDERPASSES:
// *****

VkResult
Init10RenderPasses( )
{
    HERE_I_AM( "Init10RenderPasses" );

    VkResult result = VK_SUCCESS;

    // need 2 - one for the color and one for the depth/stencil
    VkAttachmentDescription vad[2];
    //vad[0].format = VK_FORMAT_B8G8R8A8_UNORM;
    vad[0].format = VK_FORMAT_B8G8R8A8_SRGB;
    vad[0].samples = VK_SAMPLE_COUNT_1_BIT;
    vad[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    vad[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    vad[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    vad[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    vad[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    vad[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
    vad[0].flags = 0;
    //vad[0].flags = VK_ATTACHMENT_DESCRIPTION_MAT_ALIAS_BIT;

    vad[1].format = VK_FORMAT_D32_SFLOAT_S8_UINT;
    vad[1].samples = VK_SAMPLE_COUNT_1_BIT;
    vad[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    vad[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    vad[1].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    vad[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    vad[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    vad[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
    vad[1].flags = 0;

    VkAttachmentReference colorReference;
    colorReference.attachment = 0;
    colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

    VkAttachmentReference depthReference;
    depthReference.attachment = 1;
    depthReference.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

    VkSubpassDescription vsd;
    vsd.flags = 0;
    vsd.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
    vsd.inputAttachmentCount = 0;
    vsd.pInputAttachments = (VkAttachmentReference *)nullptr;
    vsd.colorAttachmentCount = 1;
    vsd.pColorAttachments = &colorReference;
    vsd.pResolveAttachments = (VkAttachmentReference *)nullptr;
    vsd.pDepthStencilAttachment = &depthReference;
    vsd.preserveAttachmentCount = 0;
    vsd.pPreserveAttachments = (uint32_t *)nullptr;

    VkRenderPassCreateInfo vrpci;
    vrpci.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
    vrpci.pNext = nullptr;
    vrpci.flags = 0;
    vrpci.attachmentCount = 2; // color and depth/stencil
    vrpci.pAttachments = vad;
    vrpci.subpassCount = 1;
    vrpci.pSubpasses = &vsd;
    vrpci.dependencyCount = 0; // ***** ERROR ?
    vrpci.pDependencies = (VkSubpassDependency *)nullptr;

    result = vkCreateRenderPass( LogicalDevice, IN &vrpci, PALLOCATOR, OUT &RenderPass );
    REPORT( "vkCreateRenderPass" );
    //vgpci.renderPass = RenderPass;

    return result;
}

```

```
// *****
// CREATE THE FRAMEBUFFERS:
// *****

VkResult
Init11Framebuffers( )
{
    HERE_I_AM( "Init11Framebuffers" );

    VkResult result = VK_SUCCESS;
    VkImageView framebufferAttachments[2];          // color + depth/stencil

    VkFramebufferCreateInfo          vfbc;
    vfbc.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    vfbc.pNext = nullptr;
    vfbc.flags = 0;
    vfbc.renderPass = RenderPass;
    vfbc.attachmentCount = 2;
    vfbc.pAttachments = framebufferAttachments;
    vfbc.width = Width;
    vfbc.height = Height;
    vfbc.layers = 1;

    framebufferAttachments[0] = PresentImageViews[0];
    framebufferAttachments[1] = DepthStencilImageView;
    result = vkCreateFramebuffer( LogicalDevice, IN &vfbc, PALLOCATOR, OUT &Framebuffers[0]
);
    REPORT( "vkCreateFramebuffer - 0" );

    framebufferAttachments[0] = PresentImageViews[1];
    framebufferAttachments[1] = DepthStencilImageView;
    result = vkCreateFramebuffer( LogicalDevice, IN &vfbc, PALLOCATOR, OUT &Framebuffers[1]
);
    REPORT( "vkCreateFramebuffer - 1" );

    return result;
}
```



```

// *****
// CREATE THE COMMAND BUFFER POOL:
// *****

// Note: need a separate command buffer for each thread!

VkResult
Init06CommandPools( )
{
    HERE_I_AM( "Init06CommandPools" );

    VkResult result = VK_SUCCESS;

    {
        VkCommandPoolCreateInfo          vcpci;
        vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
        vcpci.pNext = nullptr;
        vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
        vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );

        result = vkCreateCommandPool( LogicalDevice, IN &vcpci, PALLOCATOR, OUT &GraphicsCommandPool );
        REPORT( "vkCreateCommandPool -- Graphics" );
    }

    {
        VkCommandPoolCreateInfo          vcpci;
        vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
        vcpci.pNext = nullptr;
        vcpci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
        vcpci.queueFamilyIndex = FindQueueFamilyThatDoesTransfer( );

        result = vkCreateCommandPool( LogicalDevice, IN &vcpci, PALLOCATOR, OUT &TransferCommandPool );
        REPORT( "vkCreateCommandPool -- Transfer" );
    }

    return result;
}

// *****
// CREATE THE COMMAND BUFFERS:
// *****

VkResult
Init06CommandBuffers( )
{
    HERE_I_AM( "Init06CommandBuffers" );

    VkResult result = VK_SUCCESS;

    // allocate 2 command buffers for the double-buffered rendering:

    {
        VkCommandBufferAllocateInfo      vcbai;
        vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
        vcbai.pNext = nullptr;
        vcbai.commandPool = GraphicsCommandPool;
        vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
        vcbai.commandBufferCount = 2; // 2, because of double-buffering

        result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
        REPORT( "vkAllocateCommandBuffers - 1" );
    }

    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture

```

```
re buffer:
{
    VkCommandBufferAllocateInfo          vcbai;
    vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    vcbai.pNext = nullptr;
    vcbai.commandPool = TransferCommandPool;
    vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    vcbai.commandBufferCount = 1;

    result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &TextureCommand
Buffer );
    REPORT( "vkAllocateCommandBuffers - 2" );
}
return result;
}
```

```
// *****  
// READ A SPIR-V SHADER MODULE FROM A FILE:  
// *****  
  
VkResult  
Init12SpirvShader( std::string filename, VkShaderModule * pShaderModule )  
{  
    HERE_I_AM( "Init12SpirvShader" );  
  
    FILE *fp;  
    (void) fopen_s( &fp, filename.c_str(), "rb" );  
    if( fp == NULL )  
    {  
        fprintf( FpDebug, "Cannot open shader file '%s'\n", filename.c_str( ) );  
        return VK_SHOULD_EXIT;  
    }  
    uint32_t magic;  
    fread( &magic, 4, 1, fp );  
    if( magic != SPIRV_MAGIC )  
    {  
        fprintf( FpDebug, "Magic number for spir-v file '%s' is 0x%08x -- should be 0x%08  
x\n", filename.c_str( ), magic, SPIRV_MAGIC );  
        return VK_SHOULD_EXIT;  
    }  
  
    fseek( fp, 0L, SEEK_END );  
    int size = ftell( fp );  
    rewind( fp );  
    unsigned char *code = new unsigned char [size];  
    fread( code, size, 1, fp );  
    fclose( fp );  
  
    VkShaderModuleCreateInfo vsmci;  
    vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;  
    vsmci.pNext = nullptr;  
    vsmci.flags = 0;  
    vsmci.codeSize = size;  
    vsmci.pCode = (uint32_t *)code;  
  
    VkResult result = vkCreateShaderModule( LogicalDevice, &vsmci, PALLOCATOR, pShaderModule  
);  
    REPORT( "vkCreateShaderModule" );  
    fprintf( FpDebug, "Shader Module '%s' successfully loaded\n", filename.c_str());  
  
    delete [ ] code;  
    return result;  
}
```

```

// *****
// CREATE A DESCRIPTOR SET POOL:
// *****

VkResult
Init13DescriptorSetPool()
{
    HERE_I_AM( "Init13DescriptorSetPool" );

    VkResult result = VK_SUCCESS;

    VkDescriptorPoolSize          vdps[4];
    vdps[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[0].descriptorCount = 1;
    vdps[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[1].descriptorCount = 1;
    vdps[2].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vdps[2].descriptorCount = 1;
    vdps[3].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vdps[3].descriptorCount = 1;

#ifdef CHOICES
    VkDescriptorType:
    VK_DESCRIPTOR_TYPE_SAMPLER
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT
#endif

    VkDescriptorPoolCreateInfo          vdpci;
    vdpci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    vdpci.pNext = nullptr;
    vdpci.flags = 0;

#ifdef CHOICES
    0
    VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT
#endif

    vdpci.maxSets = 4;
    vdpci.poolSizeCount = 4;
    vdpci.pPoolSizes = &vdps[0];

    result = vkCreateDescriptorPool(LogicalDevice, IN &vdpci, PALLOCATOR, OUT &DescriptorPool);

    REPORT("vkCreateDescriptorPool");

    return result;
}

// *****
// CREATING A DESCRIPTOR SET LAYOUT:
// *****

// A DS is a set of resources bound into the pipeline as a group.
// Multiple sets can be bound at one time.
// Each set has a layout, which describes the order and type of data in that set.
// The pipeline layout consists of multiple DS layouts.

#ifdef CODE_THAT_THIS_WILL_BE_DESCRIBING
layout( std140, set = 0, binding = 0 ) uniform matrixBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 uNormalMatrix;
} Matrices;

```

```

layout( std140, set = 1, binding = 0 ) uniform lightBVuf
{
    vec4 uLightPos;
} Light;

layout( std140, set = 2, binding = 0 ) uniform miscBuf
{
    float uTime;
    int    uMode;
} Misc;

layout ( set = 3, binding = 0 ) uniform sampler2D uSampler;
#endif

VkResult
Init13DescriptorSetLayouts( )
{
    HERE_I_AM( "Init13DescriptorSetLayouts" );

    VkResult result = VK_SUCCESS;

    // arrays of >= 1 layouts:
    //DS #0:
    VkDescriptorSetLayoutBinding          MatrixSet [1];
        MatrixSet [0].binding              = 0;
        MatrixSet [0].descriptorType       = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        MatrixSet [0].descriptorCount      = 1;
        MatrixSet [0].stageFlags           = VK_SHADER_STAGE_VERTEX_BIT;
        MatrixSet [0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #1:
    VkDescriptorSetLayoutBinding          LightSet [1];
        LightSet [0].binding                = 0;
        LightSet [0].descriptorType         = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        LightSet [0].descriptorCount       = 1;
        LightSet [0].stageFlags             = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    LightSet [0].pImmutableSamplers = (VkSampler *)nullptr;

    //DS #2:
    VkDescriptorSetLayoutBinding          MiscSet [1];
        MiscSet [0].binding                  = 0;
        MiscSet [0].descriptorType           = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        MiscSet [0].descriptorCount          = 1;
        MiscSet [0].stageFlags               = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT;
    MiscSet [0].pImmutableSamplers = (VkSampler *)nullptr;

    // DS #3:
    VkDescriptorSetLayoutBinding          TexSamplerSet [1];
        TexSamplerSet [0].binding            = 0;
        TexSamplerSet [0].descriptorType     = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
        // uniform sampler2D uSampler
        // vec4 rgba = texture( uSampler, vST );
        TexSamplerSet [0].descriptorCount    = 1;
        TexSamplerSet [0].stageFlags         = VK_SHADER_STAGE_FRAGMENT_BIT;
        TexSamplerSet [0].pImmutableSamplers = (VkSampler *)nullptr;

#ifdef CHOICES
    VkDescriptorType:
    VK_DESCRIPTOR_TYPE_SAMPLER
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT

```

```

#endif

VkDescriptorSetLayoutCreateInfo          vdslc0;
vdslc0.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc0.pNext = nullptr;
vdslc0.flags = 0;
vdslc0.bindingCount = 1;
vdslc0.pBindings = &MatrixSet[0];

VkDescriptorSetLayoutCreateInfo          vdslc1;
vdslc1.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc1.pNext = nullptr;
vdslc1.flags = 0;
vdslc1.bindingCount = 1;
vdslc1.pBindings = &LightSet[0];

VkDescriptorSetLayoutCreateInfo          vdslc2;
vdslc2.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc2.pNext = nullptr;
vdslc2.flags = 0;
vdslc2.bindingCount = 1;
vdslc2.pBindings = &MiscSet[0];

VkDescriptorSetLayoutCreateInfo          vdslc3;
vdslc3.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc3.pNext = nullptr;
vdslc3.flags = 0;
vdslc3.bindingCount = 1;
vdslc3.pBindings = &TexSamplerSet[0];

result = vkCreateDescriptorSetLayout( LogicalDevice, &vdslc0, PALLOCATOR, OUT &DescriptorSetLayouts[0] );
REPORT( "vkCreateDescriptorSetLayout - 0" );

result = vkCreateDescriptorSetLayout( LogicalDevice, &vdslc1, PALLOCATOR, OUT &DescriptorSetLayouts[1] );
REPORT( "vkCreateDescriptorSetLayout - 1" );

result = vkCreateDescriptorSetLayout( LogicalDevice, &vdslc2, PALLOCATOR, OUT &DescriptorSetLayouts[2] );
REPORT( "vkCreateDescriptorSetLayout - 2" );

result = vkCreateDescriptorSetLayout( LogicalDevice, &vdslc3, PALLOCATOR, OUT &DescriptorSetLayouts[3] );
REPORT( "vkCreateDescriptorSetLayout - 3" );

return result;
}

// *****
// ALLOCATE AND WRITE DESCRIPTOR SETS:
// *****

VkResult
Init13DescriptorSets( )
{
    HERE_I_AM( "Init13DescriptorSets" );

    VkResult result = VK_SUCCESS;

    VkDescriptorSetAllocateInfo          vdsai;
    vdsai.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    vdsai.pNext = nullptr;
    vdsai.descriptorPool = DescriptorPool;
    vdsai.descriptorSetCount = 4;
    vdsai.pSetLayouts = DescriptorSetLayouts;

    result = vkAllocateDescriptorSets( LogicalDevice, IN &vdsai, OUT &DescriptorSets[0] );
    REPORT( "vkAllocateDescriptorSets" );
}

```

```
VkDescriptorBufferInfo                                vdbi0;
    vdbi0.buffer = MyMatrixUniformBuffer.buffer;
    vdbi0.offset = 0; // bytes
    vdbi0.range = sizeof(Matrices);

VkDescriptorBufferInfo                                vdbi1;
    vdbi1.buffer = MyLightUniformBuffer.buffer;
    vdbi1.offset = 0; // bytes
    vdbi1.range = sizeof(Light);

VkDescriptorBufferInfo                                vdbi2;
    vdbi2.buffer = MyMiscUniformBuffer.buffer;
    vdbi2.offset = 0; // bytes
    vdbi2.range = sizeof(Misc);

VkDescriptorImageInfo                                vdii0;
    vdii0.sampler = MyPuppyTexture.texSampler;
    vdii0.imageView = MyPuppyTexture.texImageView;
    vdii0.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

VkWriteDescriptorSet                                  vwds0;
    // ds 0:
    vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    vwds0.pNext = nullptr;
    vwds0.dstSet = DescriptorSets[0];
    vwds0.dstBinding = 0;
    vwds0.dstArrayElement = 0;
    vwds0.descriptorCount = 1;
    vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vwds0.pBufferInfo = &vdbi0;
    vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
    vwds0.pTexelBufferView = (VkBufferView *)nullptr;

    // ds 1:
VkWriteDescriptorSet                                  vwds1;
    vwds1.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    vwds1.pNext = nullptr;
    vwds1.dstSet = DescriptorSets[1];
    vwds1.dstBinding = 0;
    vwds1.dstArrayElement = 0;
    vwds1.descriptorCount = 1;
    vwds1.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vwds1.pBufferInfo = &vdbi1;
    vwds1.pImageInfo = (VkDescriptorImageInfo *)nullptr;
    vwds1.pTexelBufferView = (VkBufferView *)nullptr;

VkWriteDescriptorSet                                  vwds2;
    // ds 2:
    vwds2.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    vwds2.pNext = nullptr;
    vwds2.dstSet = DescriptorSets[2];
    vwds2.dstBinding = 0;
    vwds2.dstArrayElement = 0;
    vwds2.descriptorCount = 1;
    vwds2.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    vwds2.pBufferInfo = &vdbi2;
    vwds2.pImageInfo = (VkDescriptorImageInfo *)nullptr;
    vwds2.pTexelBufferView = (VkBufferView *)nullptr;

    // ds 3:
VkWriteDescriptorSet                                  vwds3;
    vwds3.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    vwds3.pNext = nullptr;
    vwds3.dstSet = DescriptorSets[3];
    vwds3.dstBinding = 0;
    vwds3.dstArrayElement = 0;
    vwds3.descriptorCount = 1;
    vwds3.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    vwds3.pBufferInfo = (VkDescriptorBufferInfo *)nullptr;
    vwds3.pImageInfo = &vdii0;
    vwds3.pTexelBufferView = (VkBufferView *)nullptr;

uint32_t copyCount = 0;
```

```
    // this could have been done with one call and an array of VkWriteDescriptorSets:
    vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds0, IN copyCount, (VkCopyDescriptorSet
*)nullptr );
    vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds1, IN copyCount, (VkCopyDescriptorSet
*)nullptr );
    vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds2, IN copyCount, (VkCopyDescriptorSet
*)nullptr );
    vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds3, IN copyCount, (VkCopyDescriptorSet
*)nullptr );

    return VK_SUCCESS;
}
```



```

// *****
// CREATE A PIPELINE LAYOUT:
// *****

VkResult
Init14GraphicsPipelineLayout( )
{
    HERE_I_AM( "Init14GraphicsPipelineLayout" );

    VkResult result = VK_SUCCESS;

    VkPipelineLayoutCreateInfo          vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = &DescriptorSetLayouts[0];
    vplci.pushConstantRangeCount = 0;
    vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

    result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipe
lineLayout );
    REPORT( "vkCreatePipelineLayout" );

    return result;
}

// *****
// CREATING A GRAPHICS PIPELINE:
// *****

#ifdef COMMENT
struct matBuf
{
    glm::mat4 uModelMatrix;
    glm::mat4 uViewMatrix;
    glm::mat4 uProjectionMatrix;
} Matrices;

struct lightBuf
{
    glm::vec4 uLightPos;
} Light;

struct miscBuf
{
    float uTime;
    int    uMode;
} Misc;

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

VkResult
Init14GraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader
, VkPrimitiveTopology topology, OUT VkPipeline *pGraphicsPipeline )
{
#ifdef ASSUMPTIONS
    vds[0] = VK_DYNAMIC_STATE_VIEWPORT;
    vds[1] = VK_DYNAMIC_STATE_SCISSOR;
    vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
    vprsci.depthClampEnable = VK_FALSE;
    vprsci.rasterizerDiscardEnable = VK_FALSE;

```

```

    vprsci.polygonMode = VK_POLYGON_MODE_FILL;
    vprsci.cullMode = VK_CULL_MODE_NONE; // best to do this because of the projec
tionMatrix[1][1] *= -1.;
    vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
    vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_ONE_BIT;
    vpcbas.blendEnable = VK_FALSE;
    vpcbsci.logicOpEnable = VK_FALSE;
    vpdssci.depthTestEnable = VK_TRUE;
    vpdssci.depthWriteEnable = VK_TRUE;
    vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
#endif

    HERE_I_AM( "Init14GraphicsVertexFragmentPipeline" );

    VkResult result = VK_SUCCESS;

    VkPipelineLayoutCreateInfo                                vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = DescriptorSetLayouts;
    vplci.pushConstantRangeCount = 0;
    vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

    result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipe
lineLayout );
    REPORT( "vkCreatePipelineLayout" );

    VkPipelineShaderStageCreateInfo                          vpssci[2];
    vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[0].pNext = nullptr;
    vpssci[0].flags = 0;
    vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
#ifdef BITS
    VK_SHADER_STAGE_VERTEX_BIT
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT
    VK_SHADER_STAGE_GEOMETRY_BIT
    VK_SHADER_STAGE_FRAGMENT_BIT
    VK_SHADER_STAGE_COMPUTE_BIT
    VK_SHADER_STAGE_ALL_GRAPHICS
    VK_SHADER_STAGE_ALL
#endif
    vpssci[0].module = vertexShader;
    vpssci[0].pName = "main";
    vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[1].pNext = nullptr;
    vpssci[1].flags = 0;
    vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    vpssci[1].module = fragmentShader;
    vpssci[1].pName = "main";
    vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    VkVertexInputBindingDescription                          vvibd[1]; // an array containing o
ne of these per buffer being used
    vvibd[0].binding = 0; // which binding # this is
    vvibd[0].stride = sizeof( struct vertex ); // bytes between success
ive
    vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
#ifdef CHOICES
    VK_VERTEX_INPUT_RATE_VERTEX
    VK_VERTEX_INPUT_RATE_INSTANCE
#endif
#ifdef COMMENT
    struct vertex
    {
        glm::vec3    position;
        glm::vec3    normal;
        glm::vec3    color;
        glm::vec2    texCoord;
    }

```

```

} Vertices;
#endif
    VkVertexInputAttributeDescription          vviad[4];          // an array containing one of these per vertex attribute in all bindings
    // 4 = vertex, normal, color, texture coord
    vviad[0].location = 0;          // location in the layout decoration
    vviad[0].binding = 0;          // which binding description this is part of

    vviad[0].format = VK_FORMAT_VEC3;          // x, y, z
    vviad[0].offset = offsetof( struct vertex, position );          // 0
#ifdef EXTRAS_DEFINED_AT_THE_TOP
VK_FORMAT_VEC4 = VK_FORMAT_R32G32B32A32_SFLOAT
VK_FORMAT_XYZW = VK_FORMAT_R32G32B32A32_SFLOAT
VK_FORMAT_VEC3 = VK_FORMAT_R32G32B32_SFLOAT
VK_FORMAT_STP = VK_FORMAT_R32G32B32_SFLOAT
VK_FORMAT_XYZ = VK_FORMAT_R32G32B32_SFLOAT
VK_FORMAT_VEC2 = VK_FORMAT_R32G32_SFLOAT
VK_FORMAT_ST = VK_FORMAT_R32G32_SFLOAT
VK_FORMAT_XY = VK_FORMAT_R32G32_SFLOAT
VK_FORMAT_FLOAT = VK_FORMAT_R32_SFLOAT
VK_FORMAT_S = VK_FORMAT_R32_SFLOAT
VK_FORMAT_X = VK_FORMAT_R32_SFLOAT
#endif

    vviad[1].location = 1;
    vviad[1].binding = 0;
    vviad[1].format = VK_FORMAT_VEC3;          // nx, ny, nz
    vviad[1].offset = offsetof( struct vertex, normal );          // 12

    vviad[2].location = 2;
    vviad[2].binding = 0;
    vviad[2].format = VK_FORMAT_VEC3;          // r, g, b
    vviad[2].offset = offsetof( struct vertex, color );          // 24

    vviad[3].location = 3;
    vviad[3].binding = 0;
    vviad[3].format = VK_FORMAT_VEC2;          // s, t
    vviad[3].offset = offsetof( struct vertex, texCoord );          // 36

    VkPipelineVertexInputStateCreateInfo          vpvisci;
    // used to describe the input vertex attributes
    vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
    vpvisci.pNext = nullptr;
    vpvisci.flags = 0;
    vpvisci.vertexBindingDescriptionCount = 1;
    vpvisci.pVertexBindingDescriptions = vviad;
    vpvisci.vertexAttributeDescriptionCount = 4;
    vpvisci.pVertexAttributeDescriptions = vviad;

    VkPipelineInputAssemblyStateCreateInfo          vpiasci;
    vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
    vpiasci.pNext = nullptr;
    vpiasci.flags = 0;
    vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
#ifdef CHOICES
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
#endif
    vpiasci.primitiveRestartEnable = VK_FALSE;

    VkPipelineTessellationStateCreateInfo          vptsci;
    vptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
    vptsci.pNext = nullptr;
    vptsci.flags = 0;
    vptsci.patchControlPoints = 0;          // number of patch control points

```

```

// VkPipelineGeometryStateCreateInfo          vpgsci;
// vptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
// vptsci.pNext = nullptr;
// vptsci.flags = 0;

VkViewport          vv;
vv.x = 0;
vv.y = 0;
vv.width = (float)Width;
vv.height = (float)Height;
vv.minDepth = 0.0f;
vv.maxDepth = 1.0f;

// scissoring:
VkRect2D          vr;
vr.offset.x = 0;
vr.offset.y = 0;
vr.extent.width = Width;
vr.extent.height = Height;

VkPipelineViewportStateCreateInfo          vpvsci;
vpvsci.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
vpvsci.pNext = nullptr;
vpvsci.flags = 0;
vpvsci.viewportCount = 1;
vpvsci.pViewports = &vv;
vpvsci.scissorCount = 1;
vpvsci.pScissors = &vr;

VkPipelineRasterizationStateCreateInfo          vprsci;
vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
vprsci.pNext = nullptr;
vprsci.flags = 0;
vprsci.depthClampEnable = VK_FALSE;
vprsci.rasterizerDiscardEnable = VK_FALSE;
vprsci.polygonMode = VK_POLYGON_MODE_FILL;

#ifdef CHOICES
VK_POLYGON_MODE_FILL
VK_POLYGON_MODE_LINE
VK_POLYGON_MODE_POINT
#endif
vprsci.cullMode = VK_CULL_MODE_NONE; // recommend this because of the projMat
rix[1][1] *= -1.;
#ifdef CHOICES
VK_CULL_MODE_NONE
VK_CULL_MODE_FRONT_BIT
VK_CULL_MODE_BACK_BIT
VK_CULL_MODE_FRONT_AND_BACK_BIT
#endif
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;

#ifdef CHOICES
VK_FRONT_FACE_COUNTER_CLOCKWISE
VK_FRONT_FACE_CLOCKWISE
#endif
vprsci.depthBiasEnable = VK_FALSE;
vprsci.depthBiasConstantFactor = 0.f;
vprsci.depthBiasClamp = 0.f;
vprsci.depthBiasSlopeFactor = 0.f;
vprsci.lineWidth = 1.f;

VkPipelineMultisampleStateCreateInfo          vpmsci;
vpmsci.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
vpmsci.pNext = nullptr;
vpmsci.flags = 0;
vpmsci.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
vpmsci.sampleShadingEnable = VK_FALSE;
vpmsci.minSampleShading = 0;
vpmsci.pSampleMask = (VkSampleMask *)nullptr;
vpmsci.alphaToCoverageEnable = VK_FALSE;
vpmsci.alphaToOneEnable = VK_FALSE;

VkPipelineColorBlendAttachmentState          vpcbas;
vpcbas.colorWriteMask = VK_COLOR_COMPONENT_R_BIT

```

```

VK_COLOR_COMPONENT_G_BIT
VK_COLOR_COMPONENT_B_BIT
VK_COLOR_COMPONENT_A_BIT;
vpcbas.blendEnable = VK_FALSE;
vpcbas.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
vpcbas.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
vpcbas.colorBlendOp = VK_BLEND_OP_ADD;
vpcbas.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
vpcbas.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
vpcbas.alphaBlendOp = VK_BLEND_OP_ADD;

VkPipelineColorBlendStateCreateInfo vpcbsci;
vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
vpcbsci.pNext = nullptr;
vpcbsci.flags = 0;
vpcbsci.logicOpEnable = VK_FALSE;
vpcbsci.logicOp = VK_LOGIC_OP_COPY;

#ifdef CHOICES
VK_LOGIC_OP_CLEAR
VK_LOGIC_OP_AND
VK_LOGIC_OP_AND_REVERSE
VK_LOGIC_OP_COPY
VK_LOGIC_OP_AND_INVERTED
VK_LOGIC_OP_NO_OP
VK_LOGIC_OP_XOR
VK_LOGIC_OP_OR
VK_LOGIC_OP_NOR
VK_LOGIC_OP_EQUIVALENT
VK_LOGIC_OP_INVERT
VK_LOGIC_OP_OR_REVERSE
VK_LOGIC_OP_COPY_INVERTED
VK_LOGIC_OP_OR_INVERTED
VK_LOGIC_OP_NAND
VK_LOGIC_OP_SET
#endif

vpcbsci.attachmentCount = 1;
vpcbsci.pAttachments = &vpcbas;
vpcbsci.blendConstants[0] = 0;
vpcbsci.blendConstants[1] = 0;
vpcbsci.blendConstants[2] = 0;
vpcbsci.blendConstants[3] = 0;

#ifdef EXAMPLE_OF_USING_DYNAMIC_STATE_VARIABLES
VkDynamicState vds[2];
vds[0] = VK_DYNAMIC_STATE_VIEWPORT;
vds[1] = VK_DYNAMIC_STATE_SCISSOR;
#endif

#ifdef CHOICES
VK_DYNAMIC_STATE_VIEWPORT -- vkCmdSetViewort ( )
VK_DYNAMIC_STATE_SCISSOR -- vkCmdSetScissor ( )
VK_DYNAMIC_STATE_LINE_WIDTH -- vkCmdSetLineWidth ( )
VK_DYNAMIC_STATE_DEPTH_BIAS -- vkCmdSetDepthBias ( )
VK_DYNAMIC_STATE_BLEND_CONSTANTS -- vkCmdSetBlendConstants ( )
VK_DYNAMIC_STATE_DEPTH_BOUNDS -- vkCmdSetDepthZBounds ( )
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK -- vkCmdSetStencilCompareMask ( )
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK -- vkCmdSetStencilWriteMask ( )
VK_DYNAMIC_STATE_STENCIL_REFERENCE -- vkCmdSetStencilReferences ( )
#endif
VkPipelineDynamicStateCreateInfo vpdsci;
vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdsci.pNext = nullptr;
vpdsci.flags = 0;
vpdsci.dynamicStateCount = 0; // leave turned off for now
vpdsci.pDynamicStates = (VkDynamicState *) nullptr;
#ifdef EXAMPLE_OF_USING_DYNAMIC_STATE_VARIABLES
vpdsci.dynamicStateCount = 2;
vpdsci.pDynamicStates = vds;
#endif

VkStencilOpState vsosf; // front
vsosf.failOp = VK_STENCIL_OP_KEEP;
vsosf.passOp = VK_STENCIL_OP_KEEP;
vsosf.depthFailOp = VK_STENCIL_OP_KEEP;

```

```

#ifdef CHOICES
VK_STENCIL_OP_KEEP
VK_STENCIL_OP_ZERO
VK_STENCIL_OP_REPLACE
VK_STENCIL_OP_INCREMENT_AND_CLAMP
VK_STENCIL_OP_DECREMENT_AND_CLAMP
VK_STENCIL_OP_INVERT
VK_STENCIL_OP_INCREMENT_AND_WRAP
VK_STENCIL_OP_DECREMENT_AND_WRAP
#endif

        vsosf.compareOp = VK_COMPARE_OP_NEVER;
#ifdef CHOICES
VK_COMPARE_OP_NEVER
VK_COMPARE_OP_LESS
VK_COMPARE_OP_EQUAL
VK_COMPARE_OP_LESS_OR_EQUAL
VK_COMPARE_OP_GREATER
VK_COMPARE_OP_NOT_EQUAL
VK_COMPARE_OP_GREATER_OR_EQUAL
VK_COMPARE_OP_ALWAYS
#endif

        vsosf.compareMask = ~0;
        vsosf.writeMask = ~0;
        vsosf.reference = 0;

    VkStencilOpState                                vsosb; // back
        vsosb.failOp = VK_STENCIL_OP_KEEP;
        vsosb.passOp = VK_STENCIL_OP_KEEP;
        vsosb.depthFailOp = VK_STENCIL_OP_KEEP;
        vsosb.compareOp = VK_COMPARE_OP_NEVER;
        vsosb.compareMask = ~0;
        vsosb.writeMask = ~0;
        vsosb.reference = 0;

    VkPipelineDepthStencilStateCreateInfo           vpdssci;
        vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
        vpdssci.pNext = nullptr;
        vpdssci.flags = 0;
        vpdssci.depthTestEnable = VK_TRUE;
        vpdssci.depthWriteEnable = VK_TRUE;
        vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
#ifdef CHOICES
VK_COMPARE_OP_NEVER
VK_COMPARE_OP_LESS
VK_COMPARE_OP_EQUAL
VK_COMPARE_OP_LESS_OR_EQUAL
VK_COMPARE_OP_GREATER
VK_COMPARE_OP_NOT_EQUAL
VK_COMPARE_OP_GREATER_OR_EQUAL
VK_COMPARE_OP_ALWAYS
#endif

        vpdssci.depthBoundsTestEnable = VK_FALSE;
        vpdssci.front = vsosf;
        vpdssci.back = vsosb;
        vpdssci.minDepthBounds = 0.;
        vpdssci.maxDepthBounds = 1.;
        vpdssci.stencilTestEnable = VK_FALSE;

    VkGraphicsPipelineCreateInfo                   vgpcci;
        vgpcci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
        vgpcci.pNext = nullptr;
        vgpcci.flags = 0;
#ifdef CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif

        vgpcci.stageCount = 2; // number of stages in this pipe
    line
        vgpcci.pStages = vpssci;
        vgpcci.pVertexInputState = &vpvisci;
        vgpcci.pInputAssemblyState = &vpiasci;

```

```
        vgpcci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
// &vptsci
    vgpcci.pViewportState = &vpvsci;
    vgpcci.pRasterizationState = &vprsci;
    vgpcci.pMultisampleState = &vpmsci;
    vgpcci.pDepthStencilState = &vpdssci;
    vgpcci.pColorBlendState = &vpcbsci;
    vgpcci.pDynamicState = (VkPipelineDynamicStateCreateInfo *) nullptr;
//vgpcci.pDynamicState = &vpdsci;
    vgpcci.layout = IN GraphicsPipelineLayout;
    vgpcci.renderPass = IN RenderPass;
    vgpcci.subpass = 0; // subpass number
    vgpcci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
    vgpcci.basePipelineIndex = 0;

    result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpcci, PALLOCA
TOR, OUT pGraphicsPipeline );
    REPORT( "vkCreateGraphicsPipelines" );

    return result;
}
```

```

// *****
// SETUP A COMPUTE PIPELINE:
// *****

VkResult
Init14ComputePipeline( VkShaderModule computeShader, OUT VkPipeline * pComputePipeline )
{
    HERE_I_AM( "Init14ComputePipeline" );

    VkResult result = VK_SUCCESS;

    VkPipelineShaderStageCreateInfo          vpssci;
    vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci.pNext = nullptr;
    vpssci.flags = 0;
    vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
    vpssci.module = computeShader;
    vpssci.pName = "main";
    vpssci.pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    VkPipelineLayoutCreateInfo              vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 1;
    vplci.pSetLayouts = DescriptorSetLayouts;
    vplci.pushConstantRangeCount = 0;
    vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;

    result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &ComputePipelineLayout );
    REPORT( "vkCreatePipelineLayout" );

    VkComputePipelineCreateInfo            vcpci[1];
    vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
    vcpci[0].pNext = nullptr;
    vcpci[0].flags = 0;
    vcpci[0].stage = vpssci;
    vcpci[0].layout = ComputePipelineLayout;
    vcpci[0].basePipelineHandle = VK_NULL_HANDLE;
    vcpci[0].basePipelineIndex = 0;

    result = vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, pComputePipeline );
    REPORT( "vkCreateComputePipelines" );
    return result;
}

#ifdef SAMPLE_CODE
vkBeginRenderPass( );
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipelines[0] );
vkCmdDispatch( CommandBuffer, numWGx, numWGY, numWQz );
vkEndRenderPass( );
#endif

```



```

// *****
// CREATING AND SUBMITTING THE FENCE:
// *****

VkResult
InitFence( )
{
    VkFenceCreateInfo          vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = 0;

    vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, &Fence );

    VkSubmitInfo              vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.waitSemaphoreCount = 0; // ??? uint32_t
    vsi.pWaitSemaphores = (VkSemaphore *)nullptr; // ??????
    vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr; // ??????
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = CommandBuffers;
    vsi.signalSemaphoreCount = 0; // ??????
    vsi.pSignalSemaphores = (VkSemaphore *)nullptr; // ??????

    VkResult result = vkQueueSubmit( Queue, 1, IN &vsi, IN Fence );
    // Fence can be VK_NULL_HANDLE if have no fence
    REPORT( "vkQueueSubmit" );

#ifdef SAMPLE_CODE
    result = vkWaitForFences( LogicalDevice, 1, pFences, VK_TRUE, timeout );
    REPORT( "vkWaitForFences" );
#endif

    return result;
}

// *****
// PUSH CONSTANTS:
// *****
//
// Push Constants are uniform variables in a shader.
// There is one Push Constant block per pipeline.
// Push Constants are "injected" into the pipeline.
// They are not necessarily backed by device memory, although they could be.

#ifdef COMMENT
layout( push_constant ) uniform myPushConstants_t
{
    int a;
    float b;
    int c;
} MyPushConstants;
#endif

#ifdef SAMPLE_CODE
    VkPushConstantRange          vpcr[1];
    vpcr.stageFlags = VK_SHADER_STAGE_ALL;
    vpcr.offset = 0;
    vpcr.size = << in bytes >>

    VkPipelineLayoutCreateInfo    vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = << length of array .pSetLayouts >>
    vplci.pSetLayouts = << array of type VkDescriptorSetLayout >>
    vplci.pushConstantRangeCount = << length of array pPushConstantRanges >>
    vplci.pPushConstantRanges = << array of type VkPushConstantRange >>

```

```
        result = vkCreatePipelineLayout( LogicalDevice, &vplci, PALLOCATOR, &PipelineLayout );
    #endif

    #ifdef SAMPLE_CODE
        vkCmdPushConstants( CommandBuffer, PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values );
    #endif

    // *****
    // SPECIALIZATION CONSTANTS:
    // *****
    //
    // Specialization Constants "specialize" a shader.
    // I.e., these constants get compiled-in.
    // Typically, the final code generation comes late, with calls to
    //     vkCreateComputePipelines( )
    //     vkCreateGraphicsPipelines( )
    // The compiler can make code-generation decisions based on Specialization Constants.
    // Specialization constants are good for:
    //     branching (~ #ifdef)
    //     switch
    //     loop unrolling
    //     constant folding
    //     operator simplification

    #ifdef SAMPLE_CODE
        layout( constant_id = 1 ) const bool  USE_HALF_ANGLE = true;
        layout( constant_id = 2 ) const float G = -9.8f;

        VkSpecializationMapEntry          vsme[1];
        vsme[0].constantId = << the constant_id in the layout line, uint32_t >>
        vsme[0].offset = << how far into the raw data this constant is, bytes >>
        vsme[0].size = << size of this SC in the raw data >>

        VkSpecializationInfo              vsi;
        vsi.mapEntryCount = << number of SCs to be set >>
        vsi.pmapEntries = << array of VkSpecializationMapEntry elements >>
        vsi.dataSize = << in bytes >>
        vsi.pData = << the raw data, void * >>
    #endif
#endif
```

```

// *****
// HANDLING A VULKAN ERROR RETURN:
// *****

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, "Valuidation Failed"  },
    { VK_ERROR_INVALID_SHADER_NV, "Invalid Shader"      },
    { VK_ERROR_OUT_OF_POOL_MEMORY_KHR, "Out of Pool Memory"  },
    { VK_ERROR_INVALID_EXTERNAL_HANDLE, "Invalid External Handle" },
};

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

```

```

// *****
// FENCES:
// *****

#ifdef SAMPLE_CODE
VkResult
InitFence( )
{
    VkResult result;

    VkFenceCreateInfo          vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = VK_FENCE_CREATE_SIGNALED_BIT;        // the only option

    result = vkCreateFence( LogicalDevice, IN &vfci, PALLOCATOR, &Fence );
    REPORT( "vkCreateFence" );

    result = vkGetFenceStatus( LogicalDevice, IN Fence );
#ifdef RESULT
    VK_SUCCESS:      its signaled
    VK_NOT_READY:   its not signaled
#endif
    REPORT( "vkGetFenceStatus" );

    result = vkWaitForFence( LogicalDevice, fenceCount, pFences, waitForAll, timeout );
#ifdef CHOICES
    waitForAll: VK_TRUE = wait for all fences
               : VK_FALSE = wait for any fences
    timeout    : uint64_t, timeout in nanoseconds
#endif
#ifdef RESULT
    result:     : VK_SUCCESS = returned because a fence signaled
               : VK_TIMEOUT = returned because the timeout was exceeded
#endif
    REPORT( "vkWaitForFence" );

    result = vkResetFences( LogicalDevice, count, pFences );
    REPORT( "vkResetFences" );
}
#endif

// *****
// EVENTS:
// *****
#ifdef SAMPLE_CODE
VkResult
InitEvent( )
{
    VkResult result;

    VkEventCreateInfo          veci;
    veci.sType = VK_STRUCTURE_TYPE_EVENT_CREATE_INFO;
    veci.pNext = nullptr;
    veci.flags = 0;

    VkResult result = vkCreateEvent( LogicalDevice, IN &veci, PALLOCATOR, OUT &Event );
    REPORT( "vkCreateEvent" );

    result = vkSetEvent( LogicalDevice, Event );
    REPORT( "vkSetEvent" );
    result = vkResetEvent( LogicalDevice, Event );
    REPORT( "vkResetEvent" );

    result = vkGetEventStatus( LogicalDevice, Event );
#ifdef RESULTS
    VK_EVENT_SET      : signaled
    VK_EVENT_RESET   : not signaled
#endif
}
#endif

```

```
REPORT( "vkGetEventStatus" );

result = vkCmdSetEvent( CommandBuffer, Event, pipelineStageBits );
REPORT( "vkCmdSetEvent" );
result = vkCmdResetEvent( CommandBuffer, Event, pipelineStageBits );
REPORT( "vkCmdResetEvent" );

result = vkCmdWaitEvents( CommandBuffer, eventCount, pEvents, srcPipelineStageBits, dstP
ipelineStageBits,
memoryBarrierCount, pMemoryBarriers,
bufferMemoryBarrierCount, pBufferMemoryBarriers,
imageMemoryBarrierCount, pImageMemoryBarriers );
REPORT( "vkCmdWaitEvents" );
#endif

// *****
// SEMAPHORES:
// *****

VkResult
InitSemaphore( )
{
    VkResult result = VK_SUCCESS;

    VkSemaphoreCreateInfo          vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

    result = vkCreateSemaphore( LogicalDevice, IN &vscl, PALLOCATOR, OUT &SemaphoreImageAvai
lable );
    REPORT( "vkCreateSemaphore -- image available" );

    result = vkCreateSemaphore( LogicalDevice, IN &vscl, PALLOCATOR, OUT &SemaphoreRenderFin
ished );
    REPORT( "vkCreateSemaphore -- render finished" );

    // vkQueueSubmit waits for one set of semaphores and signals another
    // Can have 2 queues, one for compute and one for graphics
    // Graphics Queue can wait on signal from Compute Queue
    // Then, Compute Queue can wait on signal from Graphics Queue

    return result;
}
```

```
VkResult
DestroyAllVulkan( )
{
    VkResult result = VK_SUCCESS;

    result = vkDeviceWaitIdle( LogicalDevice );
    REPORT( "vkWaitIdle" );

    // destroy things in the opposite order in which they were created:

    //destroy the swap chain imageViews
#ifdef TREVOR
    for(uint32_t i = 0; i < SWAPCHAINIMAGECOUNT; i++)
        vkDestroyImageView(LogicalDevice, PresentImageViews[i], PALLOCATOR);
    delete[] PresentImageViews;
    vkDestroySwapchainKHR(LogicalDevice, SwapChain, PALLOCATOR);
    vkDestroySurfaceKHR(Instance, Surface, PALLOCATOR);

    //do not need to free invdividual descriptors because we did not enable VK_DESCRIPTOR_PO
    OL_CREATE_FREE_DESCRIPTOR_SET_BIT in the flags
    vkDestroyDescriptorPool(LogicalDevice, DescriptorPool, PALLOCATOR);

    // The source code is not using the global 'Fence' variable
    //vkDestroyFence(LogicalDevice, Fence, PALLOCATOR);
    // The source code is not using the global semaphore variables
    //vkDestroySemaphore(LogicalDevice, SemaphoreImageAvailable, PALLOCATOR);

    vkFreeCommandBuffers(LogicalDevice, GraphicsCommandPool, sizeof(CommandBuffers) / sizeof
(CommandBuffers[0]), CommandBuffers);
    vkFreeCommandBuffers(LogicalDevice, TransferCommandPool, 1, &TextureCommandBuffer);

    vkDestroyRenderPass(LogicalDevice, RenderPass, PALLOCATOR);

    for(auto framebuffer: Framebuffers)
        vkDestroyFramebuffer(LogicalDevice, framebuffer, PALLOCATOR);

    vkDestroyShaderModule(LogicalDevice, ShaderModuleVertex, PALLOCATOR);
    vkDestroyShaderModule(LogicalDevice, ShaderModuleFragment, PALLOCATOR);

    //destory depth/stencil
    vkDestroyImageView(LogicalDevice, DepthStencilImageView, PALLOCATOR);
    vkDestroyImage(LogicalDevice, DepthStencilImage, PALLOCATOR);
    vkFreeMemory(LogicalDevice, DepthStencilImageMemory, PALLOCATOR);

    vkDestroyCommandPool(LogicalDevice, GraphicsCommandPool, PALLOCATOR);
    vkDestroyCommandPool(LogicalDevice, TransferCommandPool, PALLOCATOR);

    vkDestroyImageView(LogicalDevice, MyPuppyTexture.texImageView, PALLOCATOR);
    vkDestroyImage(LogicalDevice, MyPuppyTexture.texImage, PALLOCATOR);
    vkDestroySampler(LogicalDevice, MyPuppyTexture.texSampler, PALLOCATOR);
    vkFreeMemory(LogicalDevice, MyPuppyTexture.stagingMemory, PALLOCATOR);
    vkFreeMemory(LogicalDevice, MyPuppyTexture.textureMemory, PALLOCATOR);

    vkDestroyPipeline(LogicalDevice, GraphicsPipeline, PALLOCATOR);

    vkDestroyPipelineLayout(LogicalDevice, GraphicsPipelineLayout, PALLOCATOR);
    for(auto descriptorSetLayout: DescriptorSetLayouts)
        vkDestroyDescriptorSetLayout(LogicalDevice, descriptorSetLayout, PALLOCATOR);

    vkDestroyBuffer(LogicalDevice, MyVertexDataBuffer.buffer, PALLOCATOR);
    vkFreeMemory(LogicalDevice, MyVertexDataBuffer.vdm, PALLOCATOR);
    vkDestroyBuffer(LogicalDevice, MyMatrixUniformBuffer.buffer, PALLOCATOR);
    vkFreeMemory(LogicalDevice, MyMatrixUniformBuffer.vdm, PALLOCATOR);
    vkDestroyBuffer(LogicalDevice, MyLightUniformBuffer.buffer, PALLOCATOR);
    vkFreeMemory(LogicalDevice, MyLightUniformBuffer.vdm, PALLOCATOR);
    vkDestroyBuffer(LogicalDevice, MyMiscUniformBuffer.buffer, PALLOCATOR);
    vkFreeMemory(LogicalDevice, MyMiscUniformBuffer.vdm, PALLOCATOR);

    vkDestroyDevice(LogicalDevice, PALLOCATOR);
    vkDestroyInstance(Instance, PALLOCATOR);
```

```
#endif

#ifdef NOTDEF
vkFreeMemory( LogicalDevice, MyLightUniformBuffer.vdm, PALLOCATOR );
vkFreeMemory( LogicalDevice, MyMatrixUniformBuffer.vdm, PALLOCATOR );
vkFreeMemory( LogicalDevice, MyMiscUniformBuffer.vdm, PALLOCATOR );
vkFreeMemory( LogicalDevice, MyVertexDataBuffer.vdm, PALLOCATOR );
vkFreeMemory( LogicalDevice, MyPuppyTexture.vdm, PALLOCATOR );

vkDestroySemaphore( LogicalDevice, SemaphoreImageAvailable, PALLOCATOR );
vkDestroySemaphore( LogicalDevice, SemaphoreRenderFinished, PALLOCATOR );

vkDestroyBuffer( LogicalDevice, MyLightUniformBuffer.buffer, PALLOCATOR );
vkDestroyBuffer( LogicalDevice, MyMatrixUniformBuffer.buffer, PALLOCATOR );
vkDestroyBuffer( LogicalDevice, MyMiscUniformBuffer.buffer, PALLOCATOR );
vkDestroyBuffer( LogicalDevice, MyVertexDataBuffer.buffer, PALLOCATOR );

vkDestroyCommandPool( LogicalDevice, GraphicsCommandPool, PALLOCATOR );
vkDestroyCommandPool( LogicalDevice, TransferCommandPool, PALLOCATOR );

vkDestroyImage( LogicalDevice, MyPuppyTexture.texImage, PALLOCATOR );
vkDestroyImage( LogicalDevice, PresentImages[0], PALLOCATOR );
vkDestroyImage( LogicalDevice, PresentImages[1], PALLOCATOR );
vkDestroyImage( LogicalDevice, DepthStencilImage, PALLOCATOR );

vkDestroyImageView( LogicalDevice, MyPuppyTexture.texImageView, PALLOCATOR );
vkDestroyImageView( LogicalDevice, PresentImageViews[0], PALLOCATOR );
vkDestroyImageView( LogicalDevice, PresentImageViews[1], PALLOCATOR );
vkDestroyImageView( LogicalDevice, DepthStencilImageView, PALLOCATOR );

vkDestroySampler( LogicalDevice, MyPuppyTexture.texSampler, PALLOCATOR );
vkDestroyRenderPass( LogicalDevice, RenderPass, PALLOCATOR );
vkDestroyFramebuffer( LogicalDevice, Framebuffers[0], PALLOCATOR );
vkDestroyFramebuffer( LogicalDevice, Framebuffers[1], PALLOCATOR );
vkDestroyShaderModule( LogicalDevice, ShaderModuleVertex, PALLOCATOR );
vkDestroyShaderModule( LogicalDevice, ShaderModuleFragment, PALLOCATOR );

vkDestroyDescriptorPool( LogicalDevice, DescriptorPool, PALLOCATOR );
vkDestroyDescriptorSetLayout( LogicalDevice, DescriptorSetLayouts[0], PALLOCATOR );
vkDestroyDescriptorSetLayout( LogicalDevice, DescriptorSetLayouts[1], PALLOCATOR );
vkDestroyDescriptorSetLayout( LogicalDevice, DescriptorSetLayouts[2], PALLOCATOR );
vkDestroyDescriptorSetLayout( LogicalDevice, DescriptorSetLayouts[3], PALLOCATOR );

vkDestroyPipelineLayout( LogicalDevice, GraphicsPipelineLayout, PALLOCATOR );
vkDestroyPipeline( LogicalDevice, GraphicsPipeline, PALLOCATOR );

vkDestroySwapchainKHR( LogicalDevice, SwapChain, PALLOCATOR );
vkDestroySurfaceKHR( Instance, Surface, PALLOCATOR );

vkDestroyDevice( LogicalDevice, PALLOCATOR );
vkDestroyInstance( Instance, PALLOCATOR );

#endif

return result;
}

// *****
// find a bank of memory that has the right flag and the right type:
// *****

int
FindMemoryThatIsDeviceLocal( uint32_t memoryTypeBits )
{
    return FindMemoryByFlagAndType( VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, memoryTypeBits );
}

int
FindMemoryThatIsHostVisible( uint32_t memoryTypeBits )
{
    return FindMemoryByFlagAndType( VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT, memoryTypeBits );
}
```

```

int
FindMemoryByFlagAndType( VkMemoryPropertyFlagBits memoryFlagBits, uint32_t memoryTypeBits )
{
    VkPhysicalDeviceMemoryProperties vpdmp;
    vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
    for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
    {
        VkMemoryType vmt = vpdmp.memoryTypes[i];
        VkMemoryPropertyFlags vmpf = vmt.propertyFlags;
        if( ( memoryTypeBits & (1<<i) ) != 0 )
        {
            if( ( vmpf & memoryFlagBits ) != 0 )
            {
                fprintf(FpDebug, "\n***** Found given memory flag (0x%08x) and t
ype (0x%08x): i = %d *****\n", memoryFlagBits, memoryTypeBits, i );
                return i;
            }
        }
    }

    fprintf(FpDebug, "\n***** Could not find given memory flag (0x%08x) and type (0x%08x) **
***\n", memoryFlagBits, memoryTypeBits);
    throw std::runtime_error( "Could not find given memory flag and type" );
}

```

```

int
FindQueueFamilyThatDoesGraphics( )
{
    uint32_t count = -1;
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyP
roperties *)nullptr );
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );
    for( unsigned int i = 0; i < count; i++ )
    {
        if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
        {
            delete[ ] vqfp;
            return i;
        }
    }

    delete[ ] vqfp;
    return -1;
}

```

```

int
FindQueueFamilyThatDoesCompute( )
{
    uint32_t count = -1;
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyP
roperties *)nullptr );
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );
    for( unsigned int i = 0; i < count; i++ )
    {
        if( ( vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT ) != 0 )
        {
            delete[ ] vqfp;
            return i;
        }
    }

    delete[ ] vqfp;
    return -1;
}

```

```

int
FindQueueFamilyThatDoesTransfer( )

```



```
{
    uint32_t count = -1;
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyP
roperties *)nullptr );
    VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
    vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );
    for( unsigned int i = 0; i < count; i++ )
    {
        if( ( vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 )
        {
            delete[ ] vqfp;
            return i;
        }
    }

    delete[ ] vqfp;
    return -1;
}
```

```

// *****
// EXECUTE THE CODE FOR THE RENDERING OPERATION:
// *****

VkResult
RenderScene( )
{
    NumRenders++;
    if (NumRenders <= 2)
        HERE_I_AM( "RenderScene" );

    VkResult result = VK_SUCCESS;

    VkSemaphoreCreateInfo          vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore( LogicalDevice, IN &vsци, PALLOCATOR, OUT &imageReadySemaphore );

    uint32_t nextImageIndex;
    vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX,
        IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );

    //REPORT( "vkCreateSemaphore" );

    if( Verbose && NumRenders <= 2 )          fprintf(FpDebug, "nextImageIndex = %d\n", nextImageIndex);

    VkCommandBufferBeginInfo          vcbbi;
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    //vcbbi.flags = VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT;    <----- or could
use this one??
    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;

    result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
    //REPORT( "vkBeginCommandBuffer" );

    VkClearColorValue          vccv;
    vccv.float32[0] = 0.0;
    vccv.float32[1] = 0.0;
    vccv.float32[2] = 0.0;
    vccv.float32[3] = 1.0;

    VkClearDepthStencilValue          vcdsv;
    vcdsv.depth = 1.f;
    vcdsv.stencil = 0;

    VkClearValue          vcv[2];
    vcv[0].color = vccv;
    vcv[1].depthStencil = vcdsv;

    VkOffset2D o2d = { 0, 0 };
    VkExtent2D e2d = { Width, Height };
    VkRect2D r2d = { o2d, e2d };

    VkRenderPassBeginInfo          vrpbi;
    vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
    vrpbi.pNext = nullptr;
    vrpbi.renderPass = RenderPass;
    vrpbi.framebuffer = Framebuffers[ nextImageIndex ];
    vrpbi.renderArea = r2d;
    vrpbi.clearValueCount = 2;
    vrpbi.pClearValues = vcv;          // used for VK_ATTACHMENT_LOAD_OP_CLEAR
    vkCmdBeginRenderPass( CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE );

    vkCmdBindPipeline( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
}

```

```

#ifdef EXAMPLE_OF_USING_DYNAMIC_STATE_VARIABLES
    VkViewport viewport =
    {
        0., // x
        0., // y
        (float)Width,
        (float)Height,
        0., // minDepth
        1. // maxDepth
    };

    vkCmdSetViewport( CommandBuffers[nextImageIndex], 0, 1, IN &viewport ); // 0=firstViewport, 1=viewportCount

    VkRect2D scissor =
    {
        0,
        0,
        Width,
        Height
    };

    vkCmdSetScissor( CommandBuffers[nextImageIndex], 0, 1, &scissor );
#endif

    vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS
, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );

        // dynamic offset count, dynamic offsets
    //vkCmdBindPushConstants( CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values );

    // all 3 buffer for geometry:

    VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
    VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
    VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };
    VkDeviceSize offsets[1] = { 0 };

    if( UseIndexBuffer )
    {
        vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets
);
        vkCmdBindIndexBuffer( CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_
UINT32 );
    }
    else
    {
        vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets )
;
    }

    const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
    const uint32_t indexCount = sizeof(JustIndexData) / sizeof(JustIndexData[0]);
    //const uint32_t instanceCount = 1;
    const uint32_t instanceCount = NumInstances;
    const uint32_t firstVertex = 0;
    const uint32_t firstIndex = 0;
    const uint32_t firstInstance = 0;
    const uint32_t vertexOffset = 0;

    if( UseIndexBuffer )
    {
        vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, fir
stIndex, vertexOffset, firstInstance );
    }
    else
    {
        vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVert
ex, firstInstance );
    }

```

```

vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );
vkEndCommandBuffer( CommandBuffers[nextImageIndex] );

VkFenceCreateInfo          vfci;
    vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    vfci.pNext = nullptr;
    vfci.flags = 0;

VkFence renderFence;
vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence );

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
#ifdef CHOICES
VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
VK_PIPELINE_STAGE_COMMAND_PROCESS_BIT_NVX = 0x00020000,
#endif
VkQueue presentQueue;
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics( ), 0, OUT &presentQueue );

VkSubmitInfo // 0 = queueIndex
    vsi;
    vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    vsi.pNext = nullptr;
    vsi.waitSemaphoreCount = 1;
    vsi.pWaitSemaphores = &imageReadySemaphore;
    vsi.pWaitDstStageMask = &waitAtBottom;
    vsi.commandBufferCount = 1;
    vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
///
    vsi.signalSemaphoreCount = 1;
    vsi.signalSemaphoreCount = 0;
    vsi.pSignalSemaphores = &SemaphoreRenderFinished;

result = vkQueueSubmit( presentQueue, 1, IN &vsi, IN renderFence ); // 1 = submitCount
nt
if( Verbose && NumRenders <= 2 ) REPORT("vkQueueSubmit");

result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );
// waitAll, timeout
if( Verbose && NumRenders <= 2 ) REPORT("vkWaitForFences");

vkDestroyFence( LogicalDevice, renderFence, PALLOCATOR );

VkPresentInfoKHR          vpi;
    vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
    vpi.pNext = nullptr;
    vpi.waitSemaphoreCount = 0;
    vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
    vpi.swapchainCount = 1;
    vpi.pSwapchains = &SwapChain;
    vpi.pImageIndices = &nextImageIndex;
    vpi.pResults = (VkResult *)nullptr;

result = vkQueuePresentKHR( presentQueue, IN &vpi );
if( Verbose && NumRenders <= 2 ) REPORT("vkQueuePresentKHR");

vkDestroySemaphore( LogicalDevice, imageReadySemaphore, PALLOCATOR );

```

```
    return result;  
}
```

```

// *****
// RESET THE GLOBAL VARIABLES:
// *****

void
Reset( )
{
    ActiveButton = 0;
    Mode = 0;
    NeedToExit = false;
    NumInstances = 1;
    NumRenders = 0;
    Paused = false;
    Scale = 1.0;
    UseIndexBuffer = false;
    UseLighting = false;
    UseRotate = true;
    Verbose = true;
    Xrot = Yrot = 0.;

    // initialize the matrices:

    glm::vec3 eye(0.,-0.5,EYEDIST);
    glm::vec3 look(0.,-0.5,0.);
    glm::vec3 up(0.,1.,0.);
    Matrices.uModelMatrix = glm::mat4( ); // identity
    Matrices.uViewMatrix = glm::lookAt( eye, look, up );
    Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1
000. );
    Matrices.uProjectionMatrix[1][1] *= -1.;
    Matrices.uNormalMatrix = glm::mat4(glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix
)));

    // initialize the lighting information:

    Light.uKa = 0.2f;
    Light.uKd = 0.5f;
    Light.uKs = 0.3f;
    Light.uShininess = 100.f;
    Light.uEyePos = glm::vec4( eye, 1. );
    Light.uLightPos = glm::vec4( -50., -50., 10., 1. );
    Light.uLightSpecularColor = glm::vec4( 1., 1., 1., 1. );

    // initialize the misc stuff:

    Misc.uTime = 0.;
    Misc.uMode = Mode;
    Misc.uLighting = UseLighting ? 1 : 0;
}

// *****
// UPDATE THE SCENE:
// *****

void
UpdateScene( )
{
    // change the object transformation:

    if (Scale < MINSCALE)
        Scale = MINSCALE;

    if( UseRotate )
    {
        if (!Paused)
        {
            Matrices.uModelMatrix = glm::mat4(); // identity
            Matrices.uModelMatrix = glm::scale(Matrices.uModelMatrix, glm::vec3(Scale, Scale, Scale));

```

```
        const glm::vec3 axis = glm::vec3(0., 1., 0.);
        Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, (float)glm::
radians(360.f*Time / SECONDS_PER_CYCLE), axis);
    }
}
else
{
    Matrices.uModelMatrix = glm::mat4(); // identity
    Matrices.uModelMatrix = glm::scale(Matrices.uModelMatrix, glm::vec3(Scale, Scale
, Scale));
    Matrices.uModelMatrix = glm::rotate(Matrices.uModelMatrix, Yrot, glm::vec3(0., 1
., 0.));
    Matrices.uModelMatrix = glm::rotate(Matrices.uModelMatrix, Xrot, glm::vec3(1., 0
., 0.));
    // done this way, the Xrot is applied first, then the Yrot, then the Scale
}

// change the object projection:
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1
000. );
    Matrices.uProjectionMatrix[1][1] *= -1.;

// change the normal matrix:
Matrices.uNormalMatrix = glm::mat4(glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix
)));
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );

// possibly change the light position:
//Light.uKa = 0.2f;
//Light.uKd = 0.4f;
//Light.uKs = 0.4f;
//Light.uEyePos = glm::vec4( eye, 1. );
//Light.uLightPos = glm::vec4( 10., 10., 10., 1. );
//Light.uLightSpecularColor = glm::vec3( 1., 1., 1. );
//Light.uShininess = 10.f;
//Fill05DataBuffer( MyLightUniformBuffer, (void*) &Light ); // don't need this now:

// change the miscellaneous stuff:
Misc.uTime = (float)Time;
Misc.uMode = Mode;
Misc.uLighting = UseLighting ? 1 : 0;
Misc.uNumInstances = NumInstances;
Fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
}
```

```

//*****
// GLFW WINDOW FUNCTIONS:
//*****

void
InitGLFW( )
{
    glfwSetErrorCallback( GLFWErrorCallback );
    glfwInit( );

    glfwWindowHint( GLFW_CLIENT_API, GLFW_NO_API );
    glfwWindowHint( GLFW_RESIZABLE, GLFW_FALSE );
    MainWindow = glfwCreateWindow( Width, Height, "Vulkan Sample", NULL, NULL );

    uint32_t count;
    const char ** extensions = glfwGetRequiredInstanceExtensions ( &count );
    fprintf( FpDebug, "\nFound %d GLFW Required Instance Extensions:\n", count );
    for( uint32_t i = 0; i < count; i++ )
    {
        fprintf( FpDebug, "\t%s\n", extensions[ i ] );
    }

    glfwSetKeyCallback( MainWindow, GLFWKeyboard );
    glfwSetCursorPosCallback( MainWindow, GLFWMouseMotion );
    glfwSetMouseButtonCallback( MainWindow, GLFWMouseButton );
}

void
InitGLFWSurface( )
{
    VkResult result = glfwCreateWindowSurface( Instance, MainWindow, NULL, &Surface );
    REPORT( "glfwCreateWindowSurface" );
}

void
GLFWErrorCallback( int error, const char * description )
{
    fprintf( FpDebug, "GLFW Error = %d: '%s'\n", error, description );
}

void
GLFWKeyboard( GLFWwindow * window, int key, int scancode, int action, int mods )
{
    if( action == GLFW_PRESS )
    {
        switch( key )
        {
            case 'i':
            case 'I':
                UseIndexBuffer = ! UseIndexBuffer;
                break;

            case 'l':
            case 'L':
                UseLighting = ! UseLighting;
                break;

            case 'm':
            case 'M':
                Mode++;
                if( Mode >= 2 )
                    Mode = 0;
                break;

            case 'p':
            case 'P':
                Paused = ! Paused;
                break;
        }
    }
}

```



```

        case 'q':
        case 'Q':
        case GLFW_KEY_ESCAPE:
            NeedToExit = true;
            break;

        case 'r':
        case 'R':
            UseRotate = ! UseRotate;
            break;

        case 'v':
        case 'V':
            Verbose = ! Verbose;
            break;

        case '1':
        case '4':
        case '9':
            NumInstances = key - '0';
            break;

        default:
            fprintf( FpDebug, "Unknown key hit: 0x%04x = '%c'\n", key, key )
;
            fprintf( stderr, "Unknown key hit: 0x%04x = '%c'\n", key, key )
;
            fflush(FpDebug);
        }
    }

// *****
// PROCESS A MOUSE BUTTON UP OR DOWN:
// *****

void
GLFWMouseButton( GLFWwindow *window, int button, int action, int mods )
{
    if( Verbose )
        fprintf( FpDebug, "Mouse button = %d; Action = %d\n", button, ac
tion );

    int b = 0;
        // LEFT, MIDDLE, or RIGHT

    // get the proper button bit mask:
    switch( button )
    {
        case GLFW_MOUSE_BUTTON_LEFT:
            b = LEFT;
            break;

        case GLFW_MOUSE_BUTTON_MIDDLE:
            b = MIDDLE;
            break;

        case GLFW_MOUSE_BUTTON_RIGHT:
            b = RIGHT;
            break;

        default:
            b = 0;
            fprintf( FpDebug, "Unknown mouse button: %d\n", button );
    }

    // button down sets the bit, up clears the bit:
    if( action == GLFW_PRESS )
    {
        double xpos, ypos;
        glfwGetCursorPos( window, &xpos, &ypos);
        Xmouse = (int)xpos;
        Ymouse = (int)ypos;
    }
}

```

```
        ActiveButton |= b;           // set the proper bit
    }
    else
    {
        ActiveButton &= ~b;         // clear the proper bit
    }
}

// *****
// PROCESS A MOUSE MOVEMENT:
// *****

void
GLFWMouseMotion( GLFWwindow *window, double xpos, double ypos )
{
    int dx = (int)xpos - Xmouse;     // change in mouse coords
    int dy = (int)ypos - Ymouse;

    if( ( ActiveButton & LEFT ) != 0 )
    {
        Xrot += ( ANGFACT*dy );
        Yrot += ( ANGFACT*dx );
    }

    if( ( ActiveButton & MIDDLE ) != 0 )
    {
        Scale += SCLFACT * (float) ( dx - dy );

        // keep object from turning inside-out or disappearing:
        if( Scale < MINSCALE )
            Scale = MINSCALE;
    }

    Xmouse = (int)xpos;              // new current position
    Ymouse = (int)ypos;
}
```