LSU EE 4702-1 Note Set: Vulkan and the Course Library

Formatted 13:49, 5 December 2023 **David M. Koppelman**

Introduction

This note set provides some background on Vulkan and describes how to use the course library for GPU Programming at LSU. As of this writing, this set includes a brief description of Vulkan buffers and the course classes for working with them. That is followed by a description of Vulkan pipelines and the course library `VPipeline` object.

Rendering Pipeline

Generally speaking a *rendering pipeline* is a system consisting of a sequence of *stages* that process *pipeline inputs* arriving at the first stage and typically result in the writing of a frame buffer with an image corresponding to the inputs. Vulkan and OpenGL both provide APIs that allow CPU code to prepare and use rendering pipelines. The discussion here is of Vulkan and OpenGL pipelines intended for rasterization. (Both Vulkan and OpenGL also support ray tracing and compute pipelines.) Vulkan and OpenGL rendering pipelines are in some ways very similar, though the mechanisms to set them up are very different.

For the discussion below, suppose that one needs to render a scene consisting of triangles, and that the coordinates of the vertices of those triangles have been placed into an array. In some cases other arrays are provided that hold triangle vertex colors and surface normals. Here the term *vertex* will refer to all the attributes associated with a triangle vertex, in this example its coordinate, color, and surface normal. For a scene consisting of T triangles the array might consist of $3T$ vertices.

Programmable Stages and Shaders

A *shader* is a programmable stage of a rendering pipeline, the term shader can also refer to the *shader program* associated with the shader. Vulkan and OpenGL define several types of shaders. In fact there is no such thing as a generic shader. A simple rendering pipeline might have just a *vertex shader* and a *fragment shader* plus fixed-function stages. In this course we will often also consider *geometry shaders* in rendering pipelines. Compute pipelines have *compute shaders*, and ray tracing pipelines have their own set of shaders.

In OpenGL, shader programs are written in *OpenGL Shading Language (GLSL)*, a C-like language. Though some aspects of GLSL will be covered in this course, students are expected to read the GLSL 4.6 specification for additional information. In Vulkan, shader programs are written in *SPIR-V*, which is more like a compiler intermediate language (which itself resembles in some ways an assembly language). There are compilers that compile GLSL to SPIR-V, and that is what will be used in this course.

When a pipeline is created one specifies which programmable stages to include, and a shader program for each programmable stage. The mechanisms for creating a pipeline will be described elsewhere. Here the discussion will be focused on writing the shader programs themselves.

A shader is stage of a pipeline, and a defining characteristic of a pipeline is that data passes through the pipeline's stages in order. The first stage of a rendering pipeline is typically a vertex shader. A vertex shader reads one input vertex, does some computation, and then writes the result, which is also called a vertex (though it can contain different data). The input vertex consists of the attributes sent by the CPU, for the example above that would include a coordinate, color, and surface normal. The output vertex may contain different data. Each shader must define the format of its input data and output data, and the output data from one shader must match the input data expected by the next shader in the pipeline (if any) and the next fixed function stage (if any).

For further discussion of rendering pipeline see the Course Library VPipeline Object section.

Vulkan Buffers

In Vulkan, *buffer* refers to a piece of memory, perhaps not yet allocated, and information about that memory. Almost all data provided to Vulkan or retrieved from Vulkan is read or written from buffers. For example, one buffer might store the location of light sources needed by a rendering pipeline and another buffer might hold the vertex coordinates used as the inputs to the rendering pipeline. In typical use these buffers might be written on the CPU, moved to the GPU, and then used. A buffer can also be written on the GPU then moved to the CPU and read by CPU code, or a buffer can be written by one piece of GPU code (say, a compute shader), and read by another.

As mental scaffolding, one might think of a buffer as a dynamically allocated array of one or more elements. Like a dynamically allocated array, one needs to allocate a buffer before using it, and one should free it when done. But unlike storage in most languages, Vulkan requires that the intended *usage* of a buffer be specified in advance, and that the size of a buffer cannot change. Those used to languages with managed memory and lax rules, such as scripting languages like Python, might feel that Vulkan's requirements for creating and managing a buffer are absurdly complicated and place an undo burden on the programmer.
A Vulkan buffer has the following major life events:

- Creation

  Done once. Note: memory allocation is done after creation in a separate step.

- Bind buffer to device memory.

  Done once. This associates the buffer with device memory. The device memory must have already been allocated.

- Map to CPU memory.

  This is done to read or write buffer from CPU.

- Bind to a pipeline.

  When a buffer is bound to a pipeline its contents can be run by the pipeline's shaders. A buffer can be bound as a uniform or as a storage buffer.

- Destroy

  Done once, of course.

The steps of buffer creation and binding demand a substantial amount of code. No assignment in this course will require understanding and modifying this code, instead course-specific `VBuffer` classes will be used. Those who are curious about Vulkan buffer creation without the `VBuffer` classes can look at the `07_InitUniformBuffer` sample in the Vulkan-Hpp package. The code for the `VBuffer` classes in the course libraries can be found in `gp/include/vutil-buffer.h`.

Overview of Course Library Buffer Classes

The course library has several classes for managing Vulkan buffers. These classes take care of allocating and binding memory, and the details of reading and writing the buffer contents from the CPU. *Class* `VBufferV<T>` manages a buffer that can hold a single value of type `T` and has CPU storage for `T`, making it easier to read and write the value from the CPU. *Class* `VBufferVV<T>` manages storage for an array of elements of type `T`. These can be conveniently written or read from the CPUas though it were a C++ std vector. *Class* `VBuffer<T>` manages an array of elements of type `T` but without managing the CPU storage.

Declaration Example

Here are some examples of how to declare the various buffers.

```
VBufferV<float> buf_f;  // Just one float.
struct My_Struct { int a,b,c,pad4;  vec4 v; pCoor p[5]; };
VBufferV<My_Struct> buf_ms;
VBufferVV<float> buf_af; // An array of floats, with convenient CPU access.
VBuffer<float> buf_rf;   // An array of floats.
VBuffer<pCoor> buf_rc;   // An array of pCoor. (Accessed as vec4 in GLSL.)
```

These can be declared to hold any type, but one should avoid using structures for the array classes `VBufferVV<T>` and `VBuffer<T>` when being used for storage buffers, because structure access in storage buffers is inefficient.

Initialization and Usage

When a Vulkan buffer is initialized one must specify its intended usage or usages (how it will be used). It is possible to specify more than one usage, perhaps even all possible usages, but one should assume that execution will be more efficient if one only specifies the usages that are needed. Here are the common usages, shown using the Vulkan enumeration constants that signify them:

`eUniformBuffer`: The buffer will be used to hold uniforms in a pipeline. The shader code will have a corresponding `layout ( binding = X ) uniform FOO` declaration.

`eStorageBuffer`: The buffer will be used to hold storage buffer for a pipeline. `layout ( binding = X ) buffer FOO {...};` declaration.

`eVertexBuffer`: The buffer will be used to hold vertex shader inputs. (This won't be needed when using the course library since `VVertex_Buffer_Set` provides those buffers.)

`eIndexBuffer`: The buffer will be used to hold vertex shader input indices. This is needed for indexed draws.

`eRayTracingNV`: The buffer will be used to hold acceleration structures needed for ray tracing. Such buffers are managed by `VRaytrace` and should not need to be explicitly declared by user code.

The enumeration constants above can be ORed together. See the example below. Initialization of all classes is done using the `init` member function. The first argument is always `vh.qs`, something like a context, and the second argument is a bit vector of usage constants (or just one constant).

```
buf_f.init( vh.qs, vk::BufferUsageFlagBits::eUniformBuffer );
buf_mf.init( vh.qs, vk::BufferUsageFlagBits::eUniformBuffer );

// An array of floats, with convenient CPU access.
buf_af.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer );

// Allocate 100 elements.
buf_rf.init( vh.qs, vk::BufferUsageFlagBits::eStorageBuffer, 100 );
```

VBufferV Example

As an example of how a `VBufferV` might be used, consider an application that would like to provide a small set of front and back colors for shader code to use. On the CPU those colors are kept in the following structure:

```
constexpr int ncolors = 10;
struct HW03_Colors {
  pColor front[ncolors], back[ncolors];
};
```

A `VBufferV` object to hold the colors is d

Course Library VPipeline Object

The course library `VPipeline` class is used to prepare and use Vulkan rasterization pipeline. A `VPipeline` object is created from shader code, and also a specification of what the shader code will read and write, plus details on what the fixed-function hardware needs to do. That's what's needed to create the pipeline. To use it one *binds* the pipeline and its resources to what's called the *graphics pipeline bind point*, and then executes one or more *draw* commands. The VPipeline Simple Example section further below shows how all of this works.

Those reading this should understand the general concept of a rendering pipeline and the role of the rendering pipeline stages defined by Vulkan (and OpenGL), especially the vertex shader and fragment shader. Readers should also be able to write at least simple shaders in OpenGL Shading Language, and understand the difference between uniform variables and vertex shader inputs.

Rendering Pipeline Review

Recall that when a vertex shader is invoked (something like a procedure call) it reads one *vertex* (the input), which can consist of zero [sic] or more *attributes*, does some computation, and then writes one vertex, consisting of zero or more attributes (not necessarily the same as the input attributes). A vertex shader is invoked for each input vertex sent from the CPU (host).

The vertex shader can read the input attributes, and can also read *uniform variables* and *storage buffers*. The total size of uniform variables is limited, and it's a good idea to keep the size below 2 kiB. Storage buffers can be much larger.

The outputs of the vertex shader are *assembled* into primitives and used as input to the *geometry shader* if present, or the rasterizer otherwise. The type of primitive (triangle, line, point, etc) and how vertices are grouped is determined by the *primitive topology* that was set for the pipeline on the CPU. A geometry shader writes zero or more (but not too many) output primitives. The type of output primitive is specified in the shader code.

The fixed function *rasterizer* operates on the primitives emitted by the geometry shader, or if there is no geometry shader, the primitives assembled from the output of the vertex shader. For each primitive, the rasterizer determines which pixels it covers. For each pixel, a set of interpolated attributes is computed, called a *fragment*. The kind of attribute interpolation is determined by how fragment shader inputs are declared. By default, integer attributes are not interpolated, they are set to the value of the *provoking vertex*, which is usually the last vertex in the primitive. Other attributes are interpolated based on their position in eye space, by default. (For example, if the vertices have color attributes, red, green, and blue, and a fragment is in the middle of the triangle it will be white, if it is on the edge between the red and blue vertex it will be purple.) To reduce the amount of computation attributes can be interpolated by their position in clip space. The *fragment shader* is invoked on each fragment that is within the window (and which also passes what are called *early fragment tests*).

The fragment shader typically will read texture images, through an opaque handle of type `samplerXX` (say, `sampler2D`) and combine the texels with the lighted color. The result will then be written to the output attribute. Fixed functionality applies *late tests* to the fragment (the fragment shader output), and if these all pass the fixed-function *blending* (also called *frame-buffer update*) stage writes the fragment to the frame buffer.

VPipeline Overview

The `VPipeline` class helps with creating and using a Vulkan graphics rasterization pipeline. (The course library also a `VCompute` class for compute "pipelines" and a `VRaytrace` class for ray tracing "pipelines".) A `VPipeline` object is used in conjunction with one or more `VVertex_Buffer_Set` objects to manage vertex shader inputs, zero or more `VBufferV<T>` objects to manage uniform values, zero or more `VVBufferV<T>` objects to manage storage buffers, plus a `VTransform` object to manage transformations. (Samplers and texture images can also be specified.)

To create a pipeline one needs to specify the following information:

Shader Code The Vulkan API requires a compiled shader, but `VPipeline` just needs a file name plus the names of the main shader stage routines. These are provided by the `shader_code_set` member function. See the simple example below.

Vertex Shader Input Format A Vulkan pipeline needs to know the format of the vertex shader input attributes. (The input consists of a stream of vertices, each vertex consists of zero or more attributes). Vulkan expects a detailed format, such as `vk::Format::eR32G32B32A32Sfloat` (a four-element vector of floats). The `VPipeline` objects makes this considerably easier by enabling the common input types to be specified by the using the `shader_inputs_info_set<T1,T2,...>()` member function. The `Ti` are must be one of `pCoor`, `pColor`, `pNorm`, `pTCoor`, `int`, `ivec2`, `ivec4`, or `mat3x4`. Each input type specified in `shadser_inputs_info_set` is assigned a *location*, and that location must be specified in the inputs `layout` declaration. Macros `LOC_IN_POS`, `LOC_IN_COLOR`, `LOC_IN_NORMAL`, `LOC_IN_TCOOR`, `LOC_IN_INT1`, `LOC_IN_INT2`, `LOC_IN_INT4`, and `LOC_IN_ROT` are assigned to the location for these types. See the *Shader Code* step in the Simple Example below.

Uniforms Buffers and Storage Buffers A Vulkan pipeline needs information on the uniform buffer and storage buffers that will be accessed. The `VPipeline` object uses member functions starting in `ds_` to provide these. The `ds_` calls must be made before the pipeline `create` member function is called. They can be used to update the buffer at a later time. See the Simple Example below for some uses.

Primitive Topology The primitive topology must also be specified. This is specified using the `topology_set` member function.

VPipeline Simple Example

The following is an example of a simple use of a `VPipeline` object, it is based on the code in `demo-03-vulkan-one.cc`. In this example a `VPipeline` object will be set up to use shader code placed in file `demo-03-shdr.cc`. The shader code includes a vertex shader and a fragment shader. The vertex shader expects three input attributes: a coordinate (type `pCoor` on the CPU), a normal (type `pNorm` on the CPU), and a color (type `pColor` on the CPU). The shader reads two uniform values, the built-in transform uniform, and a custom uniform named `uni_light_simple` on the CPU code. (See the *Shader Code* step below for more details.)

The vertex shader inputs are managed by the `VVertex_Buffer_Set` class. It is discussed in the *Prepare Input Buffer Set* step, below. The custom uniform, `uni_light_simple`, is managed by the `VBufferV` class, and the transformation matrix is handled by `VTransform`. The `VBufferV<TYPE>` class manages a Vulkan buffer holding a value of type `TYPE`. Here, `TYPE` is `Uni_Light_Simple`, a structure holding some information about a light source. The `VTransform` class keeps track of coordinate transformations. It will be discussed at length elsewhere.

This example shows how to declare, create, use, and destroy a pipeline that will execute the shader code. (The shader code itself is in file `demo-03-shdr.cc`.)

Declare Pipeline and Supporting Objects

A `VPipeline` object along with supporting objects are declared below. The object `bset_plain` is an object which will hold the vertex shader inputs. The `VTransform` object holds the coordinate space transform and `uni_light_simple` is a buffer that holds light information.

```
// This is part of the World class in demo-03-vulkan-one.cc.
VPipeline pipe_plain;
VVertex_Buffer_Set bset_plain;
VTransform transform;
VBufferV<Uni_Light_Simple> uni_light_simple;
```

Initialize Buffers

Each `VBufferV` object must be initialized. The initialization for this example is shown below. The first argument, `vh.qs`, is a context. The second, indicates that the buffer will be used to hold uniform values.

```
uni_light_simple.init( vh.qs, vk::BufferUsageFlagBits::eUniformBuffer );
```

Create Pipeline

The code below creates the pipeline.

```
pipe_plain
  .init( vh.qs )
```

```
.ds_follow( transform )
.ds_uniform_use( "BIND_LIGHT_SIMPLE", uni_light_simple )
.shader_inputs_info_set<pCoor,pNorm,pColor>()
.shader_code_set
("demo-03-shdr-code.cc", "vs_main();", nullptr, "fs_main();")
.topology_set( vk::PrimitiveTopology::eTriangleList )
.create();
```

Pipeline creation starts with a call to `init` and ends with a call to `create`. The argument to `init`, `vh.qs`, is a context object which is provided by the course library. The member functions starting with `ds_`, here they are `ds_follow` and `ds_uniform_use`, specify that the pipeline's descriptor set will include a transform and the buffer `uni_light_simple`. The `follow` suffix in `ds_follow` indicates that the pipeline should check for changes in `transform` before each draw. The `VTransform` class is something `VPipeline` recognizes. The `ds_uniform_use` member function tells the pipeline to include a descriptor for `uni_light_simple` in the descriptor set and to put the location in `BIND_LIGHT_SIMPLE` (see the discussion of the shader code).

The `shader_code_set` line specifies the file in which the shader code can be found, and the names of the vertex, geometry, and fragment shaders' entry points. In this case entry point for the vertex shader is `vs_main()` and the entry point for the fragment shader is `fs_main()`. The `nullptr` indicates that there is no geometry shader. The `shader_inputs_info_set` indicates the types of the pipeline vertex shader inputs. The `VPipeline` object recognizes only a limited number of input types. The location for input `pCoor` is associated with location `LOC_IN_POS`, `pNorm` is associated ith location `LOC_IN_NORMAL`, and `pColor` with `LOC_IN_COLOR`. (There are several other recognized inputs.) See the descriptor of the shader code to see how the `LOC_IN_` symbols are used.

The `topology_set` member function specifies how the vertices entering the pipeline need to be grouped to form primitives. The value is a Vulkan `vk::PrimitiveTopology` enumeration constant. In this case the value indicates that a triangle will be formed with each group of three vertices. (That is, the first triangle will be formed from vertex 1, 2, and 3; the second from 4, 5, and 6; and so on.)

The create call creates the pipeline. Since creation is time consuming it should not be done more often than necessary. In the `demo-03-vulkan-one` code the pipeline is created just once but used every frame.

Shader Code

The `shader_inputs_info_set` call used in creating the pipeline specifies the input types. The vertex shader code must have matching declarations for those inputs. The data types must be compatible and the locations must be set using the appropriate `LOC_In_` symbols. The shader code for the `pipeline_plain` in this example is in `demo-03-shdr-code.cc`. The declarations for the three inputs are:

```
#ifdef _VERTEX_SHADER_
layout (location = LOC_IN_POS) in vec4 in_vertex_o;     // From pCoor
layout (location = LOC_IN_NORMAL) in vec3 in_normal_o;  // From pNorm
layout (location = LOC_IN_COLOR) in vec4 in_color;      // From pColor
```

The location for a `pCoor` must be `LOC_IN_POS` and the data type must be vec4, but the variable name, `in_vertex_o` can be whatever is useful.

Two uniforms were bound to the pipeline, one a built-in (transform), the other a user-defined uniform, `uni_light_simple`. Here is the shader code declaration corresponding to `uni_light_simple`:

```
layout ( binding = BIND_LIGHT_SIMPLE ) uniform Uni_Light
{
  vec4 pos;
  vec4 color;
} uni_light;
```

The binding must be set to the same symbol used in the `ds_uniform_use` call. The arrangement of the data items in the layout declaration must be compatible with the structure used in the CPU. The CPU structure in this case is:

```
struct Uni_Light_Simple {
  pCoor position;
```

```
  pColor color;
};
// ... further down..
  VBufferV<Uni_Light_Simple> uni_light_simple;
};
```

These two are compatible because both a `vec4` and a `pCoor` consist of four floating-point members. The names do not have to match, so it is not a problem that the shader code abbreviates the first member `pos` while the CPU code names it `position`. Obviously, one order in which the members appear must be the same in both cases.

Prepare Input Buffer Set

The `shader_inputs_info_set` call prepared the pipeline to expect each vertex to consist of attributes `pCoor`, `pNorm`, and `pColor`. It did not specify anything about where those inputs would come from. The inputs to a pipeline are conveniently managed by the `VVertex_Buffer_Set` class. The term *input buffer set* or *buffer set* will be used for refer to an instance of this class.

First, a buffer set must be reset:

```
bset_plain.reset(pipe_plain);
```

This clears any existing contents of `bset_plain`, and then sets it to expect the inputs needed by `pipe_plain`. A buffer set contains a container (standard C++ vector) for each input type. Inputs are inserted into a `VVertex_Buffer_Set` using the `<<` operator. The operator is overloaded so that inputs are placed in the appropriate container. Consider:

```
pCoor px(-4,2,-2);
bset_plain << pCoor(-2,0,-2) << pCoor(-2,2,-2) << px;
bset_plain << color_green << color_green << pColor(0,0.8,0);
pNorm snorm = cross( pCoor(-4,0,-2), pCoor(-4,2,-2), pCoor(-2,0,-2) );
bset_plain << snorm << snorm << snorm;
bset_plain << color_green << color_green << color_green;
bset_plain << pCoor(-2,0,-2) << snorm << pCoor(-2,2,-2) << snorm;
pset_plain << snorm << pCoor(-4,2,-2);
```

The second line above writes three coordinates. The first two are specified using constructors (such as `pCoor(-2,0,-2)`), the last `px`, is from a variable. They are appended to the coordinate list because all are of type `pCoor`. The next line writes three colors, and the last writes three normals. The last three lines write another set of three attributes. This times colors first and with normals and coordinates interspersed.

After the execution of the code above the three containers in `bset_plain` would have six attributes each. It is important that the number of attributes of each type must be zero or must match the number of coordinates (`pCoor`), an error message will be issued if this is not the case.

After all of the attributes are inserted into a buffer set they must be moved to the GPU. That is done by member function `to_dev`. If the coordinates or other attributes change each frame, then the buffer set must be reset and re-populated each from. But, if some part of a scene does not change, say an object that does not move, then a buffer object describing it can be prepared just once, including one call to `to_dev()`.

Record the Draw

A buffer set holds inputs to a pipeline. A pipeline is something like a piece of compiled routine that's ready to run. A pipeline is used to draw something by *recording* a *draw* command.

```
pipe_plain.record_draw(cb, bset_plain);
```

Member function `pipe_plain.record_draw(cb,bset_plain)` causes a draw command to be recorded using the buffers managed by `bset_plain` as an input. The `cb` argument is a *command buffer*, which is provided by the course library. As one might expect, this results in the inputs collected in `bset_plain` to be streamed into the shader code compiled into `pipe_plain`, resulting in the frame buffer being painted with the image described by the vertices.

But, this does not happen immediately. It will happen when the command buffer is executed, which is after, in this case, the `World::render` routine returns. That means the contents of `bset_plain` and also

`uni_light_simple` that will be used in the draw are the contents that are present when the command buffer is executed. So, if, say, `bset_plain` were reset, re-populated, and `to_dev` were called five times while in `World::render`, the only values that would be rendered are the ones set the last of those five times. The same is true for `uni_light_simple`. If one needs to render five different sets of objects, then five buffer sets would be needed, though they all could use the same pipeline.

Destroy
At some point Vulkan objects need to be destroyed. For simple pieces of code like the classroom demos, one can avoid destroying Vulkan objects and the only consequences will be the shame messages printed by the Vulkan validation layer. But, if many objects are created, used briefly, and not used again, failure to destroy them can slow and stop the system as resources run low.

The objects in the code example are destroyed by calling the `destroy` member functions:

```
uni_light_simple.destroy();
pipe_plain.destroy();
bset_plain.destroy();
transform.destroy();
```

This is done once, before the program exits. That's because the pipeline is created once and used many times.