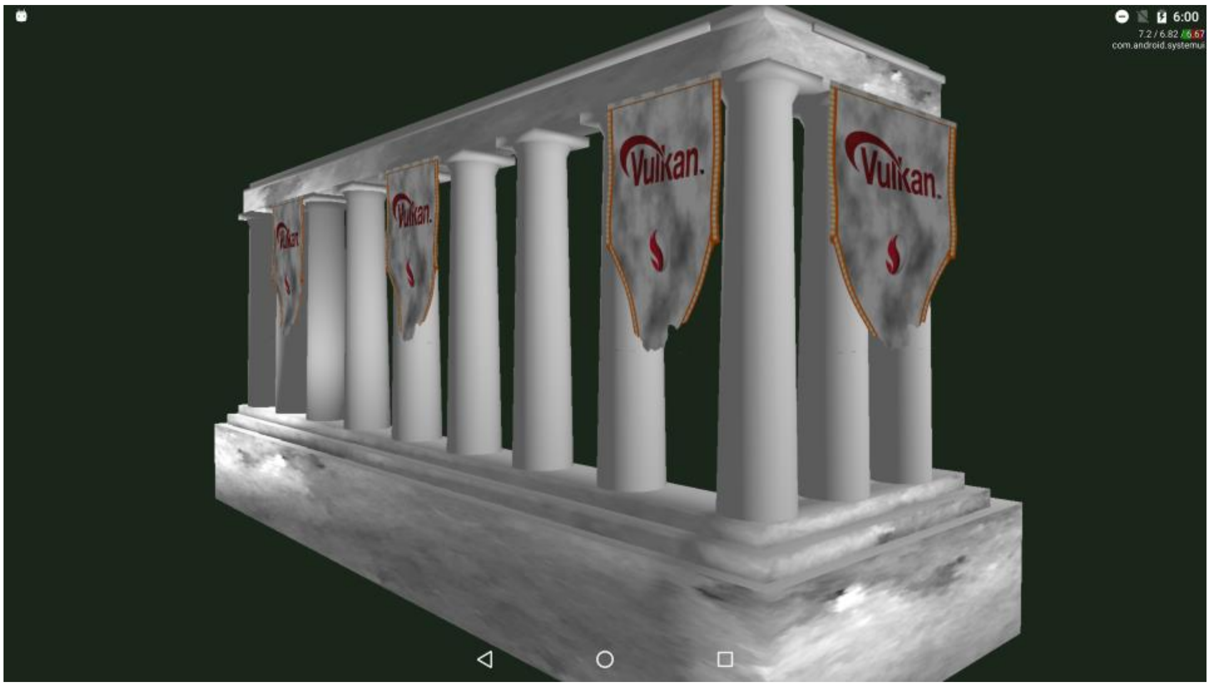
Vulkan Tutorial: Monument

Introduction

This tutorial walks though the steps required in code to implement the monument sample, which combines compute shader output with other geometry to form a scene.

Overview

In this tutorial, a simple monument mesh is rendered, followed by textured banner meshes. The banner meshes are dynamically modified each frame by a compute shader. Very simple lighting is achieved using a fragment shader. There are multiple descriptor sets for the banners, allowing each to hold their own matrices to be rendered independently in different locations.



This tutorial covers the following aspects:

- Setting up compute storage buffers.
- Using a vertex buffer as a compute shader input.
- Setting up multiple pipeline layouts, descriptor sets and uniforms for both vertex and fragment shader stages.
- Creating a compute pipeline.
- Dispatching work to a compute shader.
- Synchronization required when buffers are shared between compute and graphics pipelines.

VkSampleFramework Base Class

The `VkSampleFramework` class contains the boilerplate initialization code required to set up a Vulkan rendering environment on Android. It initializes the instance, selects the first physical device, and creates graphics and present queues.

A swapchain is created with relevant image views. The presenting and acquiring of the next swapchain image is also handled. All of these operations are explained in more detail in the 'First triangle' sample documentation.

The framework base class is abstract and expects that the `InitSample()`, `DestroySample()`, `Draw()` and `Update()` functions are implemented in the derived class.

InitSample

`InitSample()` is called after the `VkSampleFramework` has successfully completed initial setup. The derived sample classes are responsible for setting up any pipeline state, render passes, framebuffers and associated layouts and descriptors.

DestroySample

`DestroySample()` is called before `VkSampleFramework` destroys it's Vulkan object state, and should mirror InitSample with the tear-down operations for the sample.

Draw

`Draw()` should perform any per-frame actions and submit graphics operations to the device queue. For synchronization, the base class member `m_backBufferSemaphore` should be waited on to ensure any swapchain images are ready, and `m_renderCompleteSemaphore` should be signaled on completion of rendering.

Update

`Update()` is called each frame after presentation operations, and is the ideal location to update any state such as uniform data. Synchronization may need to be added, depending on the operations required.

Initialization

For this sample, we initialize:

- Textures
- Matrices and other Uniform data
- Uniform, Vertex and Storage buffers
- Layouts, Descriptor Sets and Pipelines for drawing the banners and monument mesh
- Compute Pipelines
- Command Buffers

Textures

This sample utilizes the `TextureObject` helper class. More information on this class exists in the Cornell Lights tutorial. There are two textures loaded, one for the banners, and one for the monument itself. Both textures are compressed using ASTC texture compression.

Uniforms and Storage Buffers

The sample has various uniform buffers. It has a single uniform buffer used as the vertex stage uniform for rendering the monument, and 4 other uniform buffers for each of the banners rendered on top. Additionally, there is a storage buffer defined. This storage buffer provides a time factor to the compute shader, as to affect how the banners move in the scene.

The layout of the storage buffer is as follows, and associated helper classes, is defined in the sample as follows.

```
struct ComputeBufferData
{
    float time;
};
BufferObject      m_storageBuffer;
ComputeBufferData m_storageData;
```

The BufferObject helper class creates Vulkan buffers. The buffer is initialized with the InitBuffer function specifying usage as VK_BUFFER_USAGE_STORAGE_BUFFER_BIT.

```
m_storageBuffer.InitBuffer(this, sizeof(m_storageData),
                     VK_BUFFER_USAGE_STORAGE_BUFFER_BIT, &m_storageData);
```

The InitBuffer() function of BufferObject performs the following actions:

1. `vkCreateBuffer` is called, with the relevant buffer size and buffer usage banners passed in the `VkBufferCreateInfo` structure.
2. Memory is allocated for the buffer, with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` banners set if an initial data pointer is provided.
3. If an initial data pointer is provided, the memory is mapped, and this initial data copied into the mapped memory location before being unmapped.
4. The buffer is then bound to this memory with `vkBindBufferMemory`.

The uniform buffers are now initialized, ready for referencing by descriptor sets. In the compute shader, the buffer is defined as such:

```
layout(std140, binding = 1) buffer detail
                                  {
                                    float time;
                                  };
```

Meshes, Vertex and Index Buffers

The provided MeshObject class allows reading of .obj file format meshes, and simple materials. It generates a single VertexBuffer object, containing vertex positions, normals, UV coordinates and colors. It performs triangulation into TRIANGLE_LIST format, and does not require a separate index buffer. The monument model is loaded into a VertexBuffer using this helper class.

The vertex buffer data for the banners is a grid of vertices with associated index buffer. The vertices are generated by a simple loop, adding vertices to a list. The indices are generated by triangulating quads in triangle list fashion.

The vertex buffer is initialized with the list of generated vertices, and attributes for Position, Color and UV coordinates provided. The index buffer is simply initialized, with no further details provided. We use a single `VERTEX_BUFFER_BIND_ID` in this sample for all vertex buffers, as there are no situations where multiple vertex buffers are bound at once.

```
m_surfaceVertices.InitBuffer(this, sizeof(vertex_layout) *
                        vertices.size(), VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
                        (const char*)&vertices[0]);
m_surfaceVertices.AddBinding(VERTEX_BUFFER_BIND_ID, sizeof(vertex_layout),
                             VK_VERTEX_INPUT_RATE_VERTEX);

m_surfaceVertices.AddAttribute(VERTEX_BUFFER_BIND_ID, 0, 0,
                               VK_FORMAT_R32G32B32_SFLOAT) // float3 pos
m_surfaceVertices.AddAttribute(VERTEX_BUFFER_BIND_ID, 1, sizeof(vec4).,
                            VK_FORMAT_R32G32B32A32_SFLOAT); // float4 color
m_surfaceVertices.AddAttribute(VERTEX_BUFFER_BIND_ID, 2, sizeof(vec4) * 2,
                            VK_FORMAT_R32G32_SFLOAT); // float2 uv

m_surfaceIndices.InitBuffer(this, sizeof(uint32_t) * indices.size(),
               VK_BUFFER_USAGE_INDEX_BUFFER_BIT, (const char*)&indices[0]);
```

Layouts, Descriptor Sets and Graphics Pipelines

Descriptor Sets describe the object bindings for each shader stage. There can be multiple descriptor sets, and the layout of these sets form the Pipeline Layout.

In this tutorial example, there are multiple descriptor sets, configured for the layout of two pipelines

1. A pipeline for drawing the monument model, with a vertex uniform defined containing the model, view and projection matrices, with a single fragment shader sampler binding.
2. A pipeline for drawing the banners, with a smaller vertex uniform holding a single modelViewProjection matrix, and a fragment-stage texture sampler. The vertex layout for the banner meshes is different from the monument model, so the `VkPipelineVertexInputStateCreateInfo` structure passed to `InitPipeline` is different.

Descriptor sets, their creation and their updating were covered in the Cornell Lights tutorial, and more information on how to handle them can be found in its accompanying tutorial document.

The first pipeline, for drawing the monument model, is very basic with blending disabled. The second pipeline, for drawing the banners, is created with a `VkPipelineColorBlendStateCreateInfo` object which defined a color attachment with Alpha blending enabled. The structure is created as follows.

```
    VkPipelineColorBlendAttachmentState att_state[1] = {};
    att_state[0].colorWriteMask = 0xf;
    att_state[0].blendEnable = VK_TRUE;
    att_state[0].alphaBlendOp = VK_BLEND_OP_ADD;
    att_state[0].colorBlendOp = VK_BLEND_OP_ADD;
    att_state[0].srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    att_state[0].dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    att_state[0].srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
    att_state[0].dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE;

    VkPipelineColorBlendStateCreateInfo    cb = {};
    cb.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
    cb.attachmentCount = 1;
    cb.pAttachments = &att_state[0];
```

The `cb` variable is then passed into the `VkSampleFramework::InitPipeline()` function. This function has many arguments, but most of them are optional. Passing nullptr will allow a default state to take the place for that respective creation info structure. We pass &cb to overload this, as the default disables blending.

```
InitPipeline(VK_NULL_HANDLE, 2, &shaderStages[0], m_pipelineLayout,
             m_renderPass, 0, &visci, nullptr, nullptr, nullptr,
             &rs, nullptr, nullptr, &cb, nullptr, &m_pipeline);
```

For clarity, InitPipeline is defined as follows:

```
// This version of InitPipeline requires various state, but will
// provide default versions of some Info structures if nullptr is
// passed.
void InitPipeline(
          VkPipelineCache                              pipelineCache,
          uint32_t                                     stageCount,
          const VkPipelineShaderStageCreateInfo*       stages,
          VkPipelineLayout                             layout,
          VkRenderPass                                 renderPass,
          uint32_t                                     subpass,
          const VkPipelineVertexInputStateCreateInfo*  vertexInputState,
    // info structures below can be nullptr, defaults will be provided
          const VkPipelineInputAssemblyStateCreateInfo* inputAssemblyState,
          const VkPipelineTessellationStateCreateInfo*  tessellationState,
          const VkPipelineViewportStateCreateInfo*      viewportState,
          const VkPipelineRasterizationStateCreateInfo* rasterizationState,
          const VkPipelineMultisampleStateCreateInfo*   multisampleState,
          const VkPipelineDepthStencilStateCreateInfo*  depthStencilState,
          const VkPipelineColorBlendStateCreateInfo*    colorBlendState,
          const VkPipelineDynamicStateCreateInfo*       dynState,
    // pipeline object to store created handle
          VkPipeline* pipeline);
```

Framebuffer and Render Pass

The Framebuffer objects and Render Passes are defined in the same way as the existing triangle sample, writing in to the swap chain images, with associated depth buffer.

Compute Queues

In this tutorial, we assume that the queue selected by the `VkSampleFramework` for graphics and presentation can also handle compute operations. This is checked by ensuring the physical device queue properties `queueFlags` member for the created queue family has `VK_QUEUE_COMPUTE_BIT` set. This is done in the `VkSample::InitComputeCmdBuffer()` function.

If this bit was not set, we would need to create a separate compute queue, and submit compute command buffers to that second queue. This tutorial does not go into further detail for that scenario.

Compute Pipeline

Compute pipelines are created very differently from the graphics pipelines. They still have a descriptorLayout, but can only have a single shader stage.

The layout for our compute pipeline contains two bindings.

```
    VkDescriptorSetLayoutBinding computeBindings[2] = {};

    computeBindings[0].binding = 0;
    computeBindings[0].descriptorCount = 1;
    computeBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
    computeBindings[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
    computeBindings[0].pImmutableSamplers = nullptr;

    computeBindings[1].binding = 1;
    computeBindings[1].descriptorCount = 1;
    computeBindings[1].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
    computeBindings[1].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
    computeBindings[1].pImmutableSamplers = nullptr;
```

These match up with our compute shader, and provide the mesh vertices for the banner surface as well as the time data which defines movement. The compute descriptor set is updated in the same way as others.

```
    VkWriteDescriptorSet writes[2] = {};

    VkDescriptorBufferInfo vertBufferInfo =
                                m_surfaceVertices.GetDescriptorInfo();
    writes[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    writes[0].dstBinding = 0;
    writes[0].dstSet = m_computeDescriptorSet;
    writes[0].descriptorCount = 1;
    writes[0].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
    writes[0].pBufferInfo = &vertBufferInfo;


    VkDescriptorBufferInfo detailBufferInfo =
                                  m_storageBuffer.GetDescriptorInfo();
    writes[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    writes[1].dstBinding = 1;
    writes[1].dstSet = m_computeDescriptorSet;
    writes[1].descriptorCount = 1;
    writes[1].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
    writes[1].pBufferInfo = &detailBufferInfo;

    vkUpdateDescriptorSets(m_device, 2, &writes[0], 0, nullptr);
```

The underlying buffer which contains our vertices can be passed into the `VkWriteDescriptorSet` structure for storage object bindings.

Creating the compute pipeline consists of passing the layout and a compute shader stage to `vkCreateComputePipelines`:

```
VkShaderModule sh_compute = CreateShaderModuleFromAsset("compute.spv");

VkPipelineShaderStageCreateInfo pipelineShaderStageCreateInfo = {};
pipelineShaderStageCreateInfo.sType =
                       VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
pipelineShaderStageCreateInfo.stage = VK_SHADER_STAGE_COMPUTE_BIT;
pipelineShaderStageCreateInfo.module = sh_compute;
pipelineShaderStageCreateInfo.pName = "main";

VkComputePipelineCreateInfo computePipelineCreateInfo = {};
computePipelineCreateInfo.sType =
                            VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
computePipelineCreateInfo.layout = m_computePipelineLayout;
computePipelineCreateInfo.stage = pipelineShaderStageCreateInfo;

VkResult ret = vkCreateComputePipelines(m_device, VK_NULL_HANDLE, 1,
                                        &computePipelineCreateInfo,
                                        nullptr, &m_computePipeline);
VK_CHECK(!ret);

vkDestroyShaderModule(m_device, sh_compute, nullptr);
```

Command Buffers

The command buffers are recorded once and simply re-submitted each frame. It binds the pipeline, descriptor set and vertex buffer from the mesh we're rendering. `vkCmdDraw` is used to draw the monument model. There is a set of graphics command buffers (one for each swapchain image) and a single compute command buffer.

The banner meshes are drawn with `vkCmdDrawIndexed` and an index buffer is bound with 32-bit indices.

```
    vkCmdBindIndexBuffer(cmdBuffer, m_surfaceIndices.GetBuffer(), 0,
                         VK_INDEX_TYPE_UINT32);
```

The compute command buffer is a standard primary buffer. It's make up is different to graphics commands, as there is no render pass bound. Compute shaders can only be dispatched outside of render pass scopes.

```
ret = vkBeginCommandBuffer(m_computeCmdBuffer, &cmd_buf_info);
VK_CHECK(!ret);

vkCmdBindPipeline(m_computeCmdBuffer, VK_PIPELINE_BIND_POINT_COMPUTE,
                  m_computePipeline);

vkCmdBindDescriptorSets(m_computeCmdBuffer, VK_PIPELINE_BIND_POINT_COMPUTE,
                        m_computePipelineLayout, 0, 1,
                        &m_computeDescriptorSet, 0, nullptr);

vkCmdDispatch(m_computeCmdBuffer, 16, 16, 1);

ret = vkEndCommandBuffer(m_computeCmdBuffer);
VK_CHECK(!ret);
```

The `vkCmdDispatch` function is what allows the compute shader to execute. The arguments are 16,16,1, corresponding to x,y,z dimensions of the amount of workgroups to dispatch. Our banner mesh is 64x64 vertices, and our workgroup size is defined in the compute shader as 4x4x1, therefore we dispatch 16 workgroups in the x and y dimension.

Per Frame Actions

Draw

The draw function submits to the queue of our device both the compute shader dispatch command buffer and the graphics command buffer associated with the current swap chain index.

We use `m_backBufferSemaphore` to wait on the back buffer being ready for use, and also signal `m_renderCompleteSemaphore` on the submitted buffers completing execution, allowing for presentation of the buffer to the surface at the appropriate time. Presentation is handled by the `VkSampleFramework` base class.

In this sample, the command buffers are submitted with one `vkQueueSubmit` call.

```
    VkCommandBuffer buffers[2] = {
            m_computeCmdBuffer,
            m_swapchainBuffers[m_swapchainCurrentIdx].cmdBuffer,
    };

    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.pNext = nullptr;
    submitInfo.waitSemaphoreCount = 1;
    submitInfo.pWaitSemaphores = &m_backBufferSemaphore;
    submitInfo.pWaitDstStageMask = nullptr;
    submitInfo.commandBufferCount = 2;
    submitInfo.pCommandBuffers = &buffers[0];
    submitInfo.signalSemaphoreCount = 1;
    submitInfo.pSignalSemaphores = &m_renderCompleteSemaphore;

    VkResult err;
    err = vkQueueSubmit(m_queue, 1, &submitInfo,  VK_NULL_HANDLE);
    VK_CHECK(!err);
```

Buffer Synchronization

As the rendering commands depend on the vertices which are output from the compute shader, we need to ensure the buffer is not being written to by the compute shader when it is being read by the graphics pipeline in the vertex shader.

To do this, we use a Buffer Memory Barrier, initiated on the graphics command buffer. This will create a barrier so that compute shader stage writes have completed before vertex attributes are read. The barrier is created using a `VkBufferMemoryBarrier` structure.

```
// This barrier ensures compute shader writes to our vertex buffer are
complete before
// we use the buffer in rendering
VkBufferMemoryBarrier bufferMemoryBarrier = {};
bufferMemoryBarrier.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
bufferMemoryBarrier.pNext = nullptr;
bufferMemoryBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
bufferMemoryBarrier.dstAccessMask = VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT;
bufferMemoryBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
bufferMemoryBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
bufferMemoryBarrier.buffer = m_surfaceVertices.GetBuffer();
bufferMemoryBarrier.offset = 0;
bufferMemoryBarrier.size = m_surfaceVertices.GetAllocSize();

vkCmdPipelineBarrier(cmdBuffer, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
                     VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
                     0, 0, nullptr, 1, &bufferMemoryBarrier, 0, nullptr);
```

The `bufferMemoryBarrier` structure specifies the `m_surfaceVertices` buffer and allocation size. It also specifies the access masks required. The `vkCmdPipelineBarrier` function specifies the pipeline stages.

More information about the various access masks and pipeline stages can be found in the Vulkan Specification document.

Update

The update function is called each frame allowing for modification of uniform buffers and other state.

The only uniform which is updated is the storage buffer associated with the compute shader, as this allows for the creation of the ripple effect the banners have. It is the same method as for uniform buffers in the Cornell Lights tutorial. Map the memory, write the memory and then unmap the memory. The memory must not be mapped when the device is using it, and the `Update()` function is only called by `VkSampleFramework` after a `vkDeviceWaitIdle` function call.

```
ret = vkMapMemory(m_device, m_storageBuffer.GetDeviceMemory(), 0,
                  m_storageBuffer.GetAllocSize(), 0, (void **) &pData);
assert(!ret);

m_storageData.time = time;

memcpy(pData, &m_storageData, sizeof(m_storageData));

vkUnmapMemory(m_device, m_storageBuffer.GetDeviceMemory());
```

Teardown

`DestroySample()` is called by the `VkSampleFramework` base class when a teardown is required. We only need to destroy our created state, which is the vertex, mesh and buffers assets, as well as pipeline, associated layouts and descriptor sets.

The Compute Shader

The compute shader takes as input an array of vertices and a time factor. It uses the time value as well as the vertex offsets in a 64x64 grid to move the vertex position in a single axis, to simulate a ripple effect. The top part of the mesh is fixed, and the intensity of ripple increases as the vertex Y coordinate increases towards the bottom of the mesh.

In this example, we use a 4x4 work-group size. This means we submit to `vkCmdDispatch` 16 workgroups in both dimension to cover the 64x64 grid.

```
/*
 * Compute shader
 */
#version 450
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

layout (local_size_x =4, local_size_y = 4) in;

struct vertex_layout {
        vec4 pos;
        vec4 color;
        vec4 uv;
    };
```

```
layout(std140, binding = 0) buffer data
                                    {
                                       vertex_layout vertices[];
                                    };


layout(std140, binding = 1) buffer detail
                                    {
                                      float time;
                                    };

void main()
{
  // 64x64 grid
  uint loc = gl_GlobalInvocationID.y * 64 + gl_GlobalInvocationID.x;

  float pi = 3.14f;
  float factor = time + gl_GlobalInvocationID.y*pi*4.0f;
  float intense = (64-gl_GlobalInvocationID.y)/64.0f;
  float offset = sin(factor*0.015f);

  vertices[loc].pos.y =  offset*0.25f*intense;
}
```