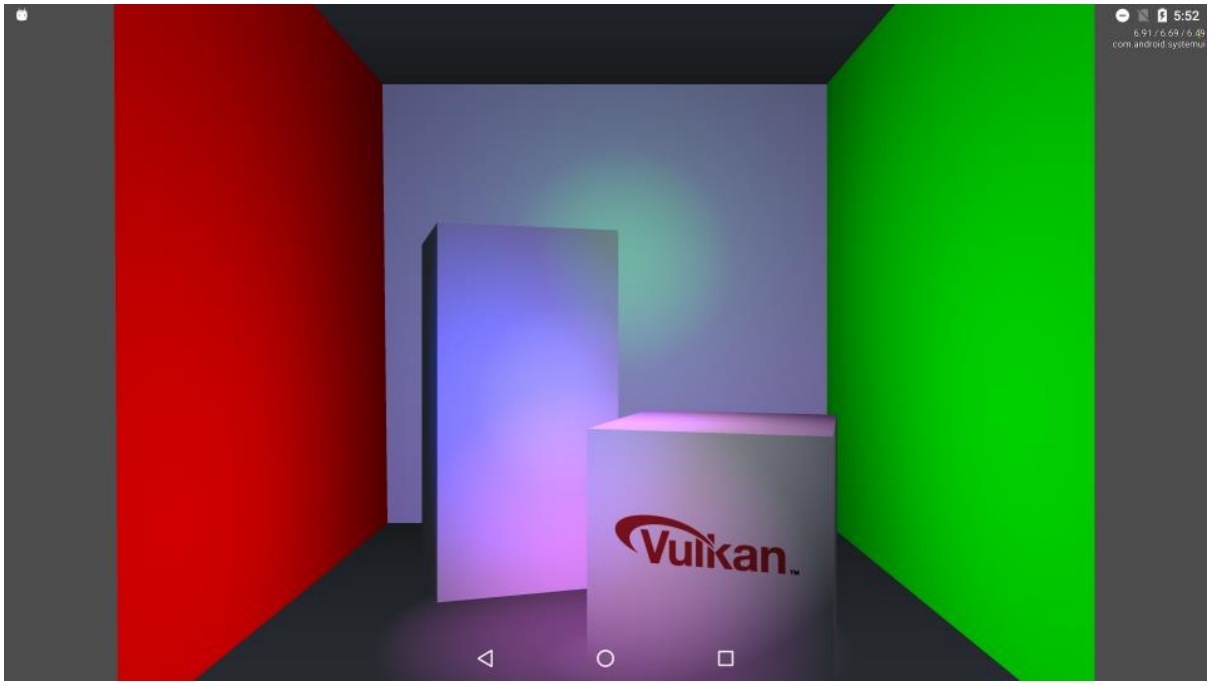
Vulkan Tutorial: Cornell Lights

Introduction

This tutorial walks though the steps required in code to implement the Cornell Lights sample, in which a Cornell Box scene is rendered using a single vertex buffer, with simple colored point lights moving around the scene.

Overview

In this tutorial, the lights are implemented using a fragment shader that iteratively adds brightness and diffuse coloring depending on light positions and colors in world space. The fragment shader has a uniform containing a number of point light locations and colors. There is also a texture sampler, and this texture is mapped to a side of the small box using correct texture coordinates in the mesh object.



This tutorial covers the following aspects:

- Loading a texture in the simplest way, with no mip-mapping.
- Setting up a Vertex Buffer with the required binding and attributes.
- How to set up buffers containing uniforms, and show their representation in the GLSL shaders.
- Setting up the Pipeline Layouts, Descriptor Sets and Uniforms for both vertex and fragment shader stages.

- Explaining how the vertex buffers and descriptor sets are submitted to command buffers.
- Show how to modify the contents of the uniforms.

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 a Texture, Vertex buffer, and Uniform buffer first.

Textures

This sample utilizes the `TextureObject` helper class. This `FromTGAFile()` function of this class allows creating a texture from a TGA image file. The texture itself in this example is created lazily, with no mipmaps. The series of steps to generate a texture is as follows:

1. Using the Android Asset Manager, load the image file and obtain the image data using `LoadTGAFromMemory`.
2. Create a `VkImage` object, with `VK_IMAGE_TYPE_2D` as the image type, the format as `VK_FORMAT_R8G8B8A8_UNORM`, the width and height specified in the .extent member, mipLevels and arrayLayers set to 1, tiling set to `VK_IMAGE_TILING_LINEAR` and usage `VK_IMAGE_USAGE_SAMPLED_BIT`.
3. Use `vkGetImageMemoryRequirements` to populate a `VkMemoryRequirements` structure with relevant details about the newly created `VkImage` object.
4. Using the memory requirements, allocate device memory, and bind that memory to the image using `vkBindImageMemory`.
5. Map the memory so that it can be written to from the CPU. Copy the image data from the loaded TGA file into the mapped memory as `VK_FORMAT_R8G8B8A8_UNORM`, using details like image `rowPitch` obtained by calling `vkGetImageSubresourceLayout`. Once the memory has been modified, unmap the memory.
6. Create a sampler for the image, with relevant UV modes and min/mag filters, such as `VK_FILTER_LINEAR`.
7. Create an image view into the image with `vkCreateImageView`.

The TextureObject class holds the sampler, image, view and memory objects internally and exposes accessors for samples to use.

Uniforms

The sample has two uniform buffers – a vertex stage uniform, containing matrixes, and a fragment stage uniform, containing a list of lights.

The layout of these uniforms are as follows.

```
struct LightConstants
 {
     glm::vec4 posAndSize[4];
     glm::vec4 color[4];
     int numLights;
 };
 LightConstants m_lights;

 struct VertexUniform
 {
     glm::mat4 modelViewProjection;
     glm::mat4 model;
     glm::mat4 view;
     glm::vec3 camPos;
 };
 VertexUniform  m_vertUniformData;
```

The BufferObject helper class creates Vulkan buffers. This allows the uniform buffers to be created easily:

```
void VkSample::InitUniformBuffers()
{
    m_lightsUniformBuffer.InitBuffer(this, sizeof(m_lights),
                  VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, &m_lights);
    m_vertShaderUniformBuffer.InitBuffer(this, sizeof(m_vertUniformData),
                  VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, &m_vertUniformData);
}
```

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

1. `vkCreateBuffer` is called, with the relevant buffer size and buffer usage flags passed in the `VkBufferCreateInfo` structure.
2. Memory is allocated for the buffer, with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` flags 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 fragment shader, the lights uniform is defined as a block with the set, binding, and standard:

```
layout(std140, set = 0, binding = 2) uniform lightDefns {
      vec4 posSize[4];
      vec4 color[4];
      int numLights;
} lights;
```

Meshes

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 VertexBuffer object internal to the MeshObject is a derived class of BufferObject, containing vertex buffer bindings and attribute details. This extra data is required by the vertex input stage of the graphics pipeline, to describe the layout of the vertex buffer data. Given the vertex layout as this C structure:

```
   struct vertex_layout {
        float pos[3];
        float color[4];
        float uv[2];
        float normal[3];
        float binormal[3];
        float tangent[4];
    };
```

The vertex attributes and bindings are as follows:

```
    meshObject->m_vertexBuffer.AddBinding(binding, sizeof(vertex_layout),
```

```
                                                VK_VERTEX_INPUT_RATE_VERTEX);
    meshObject->m_vertexBuffer.AddAttribute(binding, 0, 0,
                        VK_FORMAT_R32G32B32_SFLOAT);    // float3 position
    meshObject->m_vertexBuffer.AddAttribute(binding, 1, sizeof(float) * 3,
                        VK_FORMAT_R32G32B32A32_SFLOAT); // float4 color
    meshObject->m_vertexBuffer.AddAttribute(binding, 2, sizeof(float) * 7,
                        VK_FORMAT_R32G32_SFLOAT);       // float2 uv
    meshObject->m_vertexBuffer.AddAttribute(binding, 3, sizeof(float) * 9,
                        VK_FORMAT_R32G32B32_SFLOAT);    // float3 normal
    meshObject->m_vertexBuffer.AddAttribute(binding, 4, sizeof(float) * 12,
                        VK_FORMAT_R32G32B32_SFLOAT);   // float3 binormal
    meshObject->m_vertexBuffer.AddAttribute(binding, 5, sizeof(float) * 15,
                        VK_FORMAT_R32G32B32_SFLOAT);   // float3 tangent
```

The definition of AddAttribute is as follows:

```
void AddAttribute(uint32_t binding, uint32_t location, uint32_t offset,
                  VkFormat format);
```

And defining the locations and types as above exposes the following vertex attribute inputs in the vertex shader:

```
layout (location = 0) in vec3 vertPos;
layout (location = 1) in vec4 vertColor;
layout (location = 2) in vec2 vertUV;
layout (location = 3) in vec3 vertNormal;
layout (location = 4) in vec3 vertbinormal;
layout (location = 5) in vec3 verttangent;
```

The `VertexBufferObject` contains a function `CreatePipelineState()` which returns a `VkPipelineVertexInputStateCreateInfo` structure which can be passed directly to the pipeline create functions, to allow this mesh to be used.

Descriptor Sets

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

In this tutorial example, there is a single descriptor set. It contains 3 bindings:

1. A vertex shader stage uniform for the model, view and projection matrices,
2. A fragment shader stage sampler - the Vulkan logo texture,
3. A fragment shader stage uniform containing light definitions.

We create a descriptor set layout for this, by passing an array of descriptors to the initialization function. This example is as follows:

```
    // This sample has many bindings: 1 sampler and two uniforms
    VkDescriptorSetLayoutBinding uniformAndSamplerBinding[4] = {};
    // Our MVP matrix
```

```
    uniformAndSamplerBinding[0].binding = 0;
    uniformAndSamplerBinding[0].descriptorCount = 1;
    uniformAndSamplerBinding[0].descriptorType =
                                VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC;
    uniformAndSamplerBinding[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
    uniformAndSamplerBinding[0].pImmutableSamplers = nullptr;
    // Our texture sampler
    uniformAndSamplerBinding[1].binding = 1;
    uniformAndSamplerBinding[1].descriptorCount = 1;
    uniformAndSamplerBinding[1].descriptorType =
                                VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    uniformAndSamplerBinding[1].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
    uniformAndSamplerBinding[1].pImmutableSamplers = nullptr;
    //lights
    uniformAndSamplerBinding[2].binding = 2;
    uniformAndSamplerBinding[2].descriptorCount = 1;
    uniformAndSamplerBinding[2].descriptorType =
                                VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC;
    uniformAndSamplerBinding[2].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
    uniformAndSamplerBinding[2].pImmutableSamplers = nullptr;

    VkDescriptorSetLayoutCreateInfo descriptorSetLayoutCreateInfo = {};
    descriptorSetLayoutCreateInfo.sType =
                      VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    descriptorSetLayoutCreateInfo.pNext = nullptr;
    descriptorSetLayoutCreateInfo.bindingCount = 3;
    descriptorSetLayoutCreateInfo.pBindings = &uniformAndSamplerBinding[0];

    ret = vkCreateDescriptorSetLayout(m_device,
             &descriptorSetLayoutCreateInfo, nullptr, &m_descriptorLayout);
```

Note that at this point, only the type of descriptor, the binding number, and what shader stage they apply to is defined. We need to now create a descriptor set according to this layout. Once the descriptor set is created, to bind our uniform buffer and texture samplers we perform a write operation.

```
    VkDescriptorSetAllocateInfo descriptorSetAllocateInfo = {};
    descriptorSetAllocateInfo.sType =
                           VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    descriptorSetAllocateInfo.pNext = nullptr;
    descriptorSetAllocateInfo.descriptorPool = m_descriptorPool;
    descriptorSetAllocateInfo.descriptorSetCount = 1;
    descriptorSetAllocateInfo.pSetLayouts = &m_descriptorLayout;

    err = vkAllocateDescriptorSets(m_device, &descriptorSetAllocateInfo,
                                   &m_descriptorSet);
    VK_CHECK(!err);

    VkDescriptorImageInfo descriptorImageInfo = {};

    descriptorImageInfo.sampler = m_texVkLogo.GetSampler();
    descriptorImageInfo.imageView = m_texVkLogo.GetView();
    descriptorImageInfo.imageLayout = m_texVkLogo.GetLayout();
```

```
    VkWriteDescriptorSet writes[3] = {};

    VkDescriptorBufferInfo vertUniform =
                        m_vertShaderUniformBuffer.GetDescriptorInfo();
    writes[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    writes[0].dstBinding = 0;
    writes[0].dstSet = m_descriptorSet;
    writes[0].descriptorCount = 1;
    writes[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC;
    writes[0].pBufferInfo = &vertUniform;

    // diffuse
    writes[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    writes[1].dstBinding = 1;
    writes[1].dstSet = m_descriptorSet;
    writes[1].descriptorCount = 1;
    writes[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    writes[1].pImageInfo = &descriptorImageInfo;

    // lights
    VkDescriptorBufferInfo lightsInfo =
                                m_lightsUniformBuffer.GetDescriptorInfo();
    writes[2].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    writes[2].dstBinding = 2;
    writes[2].dstSet = m_descriptorSet;
    writes[2].descriptorCount = 1;
    writes[2].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC;
    writes[2].pBufferInfo = &lightsInfo;

    vkUpdateDescriptorSets(m_device, 3, &writes[0], 0, nullptr);
```

At this point, the descriptor set now holds valid references to our texture sampler and uniform buffers. It can be successfully bound in the pipeline when submitting draw calls via command buffers.

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.

Pipeline

The pipeline is initialized in the same fashion as the first triangle sample, except that it uses the Pipeline Layout which is defined with the new Descriptor Set Layout we require. The vertex input details required for pipeline creation can be generated by calling the `CreatePipelineState()` function on the `VertexBuffer` object contained in our scene mesh, `m_mesh`.

```
    VkPipelineVertexInputStateCreateInfo visci =
                                    m_mesh.Buffer().CreatePipelineState();
```

The `VkSampleFramework` has a helper function for creating basic pipelines, and takes the input states as well as vertex and fragment `VkShaderModule` objects. The shaders themselves are pretty standard, the vertex shader multiplying vertex positions by the modelViewProjection matrix, and various vertex attributes being passed to the fragment shader.

Command Buffer

The command buffers are recorded once and simply re-submitted each frame. It simply binds the pipeline, descriptor set and vertex buffer from the mesh we're rendering. `vkCmdDraw` is used to draw the scene.

```
        vkCmdBeginRenderPass(cmdBuffer, &rp_begin,
                             VK_SUBPASS_CONTENTS_INLINE);

        // Set our pipeline. This holds all major state
        // the pipeline defines, for example, that the vertex buffer is a
        // triangle list.
        vkCmdBindPipeline(cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
                          m_pipeline);

        //bind out descriptor set, which handles our uniforms and samplers
        vkCmdBindDescriptorSets(cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
                        m_pipelineLayout, 0, 1, &m_descriptorSet, 0, NULL);

        // Bind our vertex buffer, with a 0 offset.
        VkDeviceSize offsets[1] = {0};
        vkCmdBindVertexBuffers(cmdBuffer, VERTEX_BUFFER_BIND_ID, 1,
                               &m_mesh.Buffer().GetBuffer(), offsets);

        // Issue a draw command
        vkCmdDraw(cmdBuffer, m_mesh.GetNumVertices(), 1, 0, 0);

        // Now our render pass has ended.
        vkCmdEndRenderPass(cmdBuffer);
```

Per Frame Actions

Draw

The draw function simply submits to the graphics queue of our device the 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.

Update

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

Updating the lights buffer is done by first ensuring the light positions are correct for this frame. This example simply uses a globally increasing number to use as a base for modifying light positions with trigonometric functions. Once the light locations and colors are updated, we update the data in the uniform buffer by mapping the device memory.

```
    m_lights.posAndSize[0].x = sinf(spinAngle*0.2f)*1.0f;
    m_lights.posAndSize[0].y = -0.5f + cosf(2.0f+ spinAngle*0.1f)*1.0f;
    m_lights.posAndSize[0].z = 1.0f;
    m_lights.posAndSize[0].w = 2.2f;

    // …

    ret = vkMapMemory(m_device, m_lightsUniformBuffer.GetDeviceMemory(), 0,
                  m_lightsUniformBuffer.GetAllocSize(), 0, (void**)&pData);

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

    vkUnmapMemory(m_device, m_lightsUniformBuffer.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 pipeline, associated layouts and descriptor sets.

Additionally, our mesh gets destroyed with a call to `.Destroy()`. The uniform buffers and texture helper classes have the same `Destroy()` function.