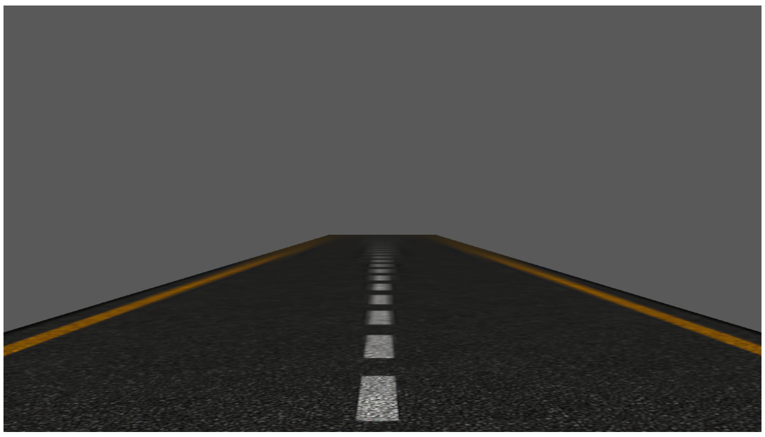
COMP5822M – Exercise 1.4
Buffers and Textures

Contents



Exercise 4 introduces Vulkan data resources, specifically buffers and textures. While textures are mainly used for texturing as their name implies, the buffers serve dual purposes. In Exercise 4, they are used to store vertex data (i.e., as vertex buffers) and uniform data (i.e., as uniform buffers). Textures and uniform buffers form uniform inputs to the graphics pipeline. Uniform inputs are declared via *descriptors*, which are further grouped into one or more *descriptor sets*. All pipelines, including graphics and compute, use this mechanism. Vertex buffers form the per-vertex input attributes. Per-vertex attributes are unique to the graphics pipeline and consequently directly declared as part of it. Refer to Figure 1 for an overview.

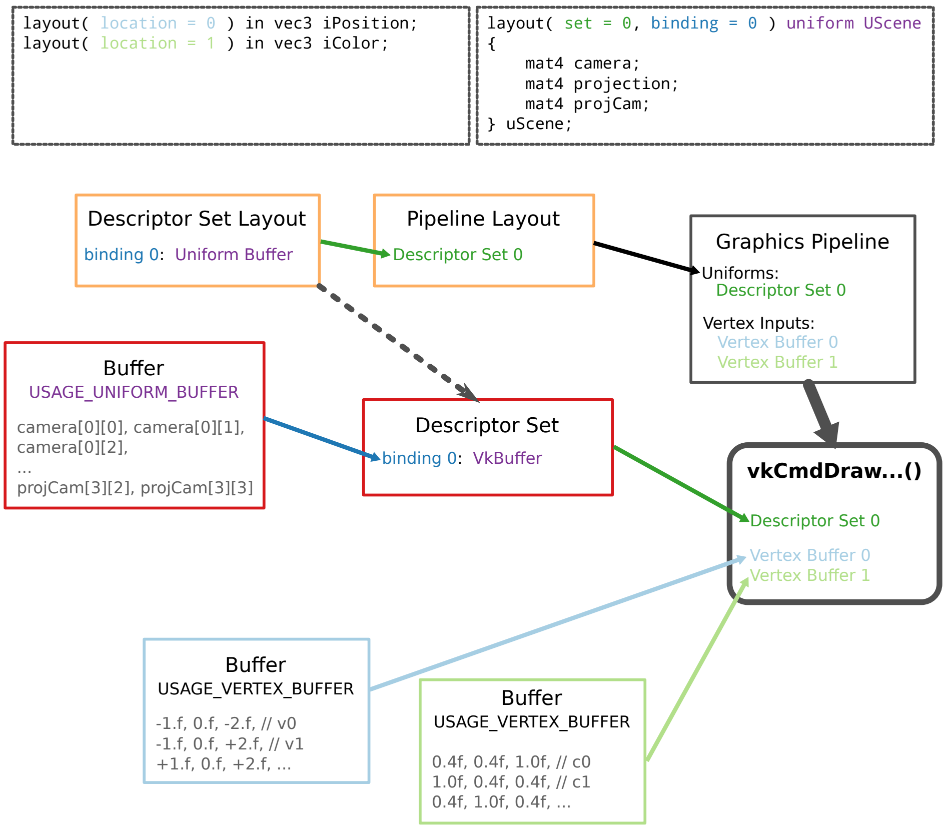
Exercise 4 first presents the vertex buffers to store the vertex data, which was previously hardcoded in the shaders. Next, Exercise 4 adds uniform buffers (Vulkan) and the corresponding uniform interface blocks (GLSL) to pass a projection matrix to the shader. With this, Exercise 4 finally starts drawing in 3D ☺. Ultimately, Exercise 4 introduces textures to make the renderings more detailed and interesting.

1 Project Updates

As always, grab the exercise code and browse through it. You will be working with the code in the `exercise4` directory and with some of the code in the `labutils` project. Transfer your `labutils` solutions from Exercise 3 to Exercise 4. Exercise 4 adds a few new files to `labutils`, but the code covered in previous exercises is otherwise unchanged.

Third Party Software Exercise 4 introduces two additional third-party libraries:

- GLM, a header only library defining common matrix and vector types, as well as associated operations/functions. It also defines a few helpers, for example for creating projection matrices and other special values. While originally designed with OpenGL in mind, the library has been updated with functionality related to Vulkan and other APIs.
- Vulkan Memory Allocator (VMA), a single-source library originally developed by AMD that helps with Vulkan memory management. Aside from providing convenience functions to allocate buffers and images, the library allocates Vulkan memory in large chunks and suballocates from these for individual resources. This is the recommended/best practice for memory allocations.



```
layout( location = 0 ) in vec3 iPosition;
layout( location = 1 ) in vec3 iColor;
```

```
layout( set = 0, binding = 0 ) uniform UScene
{
    mat4 camera;
    mat4 projection;
    mat4 projCam;
} uScene;
```

Figure 1: *Overview of how data is fed into a draw call. The layouts (orange outline) declare what type of uniform data is fed into a graphics pipeline. Multiple graphics pipelines may utilize the same layouts. The actual data is specified when drawing. Uniform data is bound via descriptor sets (red); vertex buffers (light green and light blue) are bound as-is. The overview corresponds to the state after Section 3. Later sections add additional data elements, specifically textures.*

Labutils You will again find a few new files in `labutils`:

`allocator.{hpp,cpp}` C++ wrapper around the VMA allocator object (`VmaAllocator`). It is described in Section 2.

`angle.hpp` A few convenience functions related to angles (e.g. conversion between degrees and radians).

`vkbuffer.{hpp,cpp}` Wrapper for Vulkan buffers. This is similar to the buffer object introduced in Exericse 2, but uses a `VmaAllocation` instead of the raw Vulkan `VkDeviceMemory` allocation. Introduced in detail in Section 2.

`vkimage.{hpp,cpp}` Wrapper for Vulkan images, similar to the buffer object described above. See Section 4 for details.

2 Vertex Buffers

(Vulkan Memory Allocator (VMA) — Buffer utilities— Buffer barrier — Vertex Data — Graphics Pipeline — Drawing)

So far, the exercises have generated the geometry directly in the vertex shader. While this can be very efficient when it is possible, we frequently need to render models and meshes modelled in dedicated software. To do so, the vertex data defining the meshes is stored in memory, specifically in `VkBuffer`s. When executing, the graphics pipeline reads the per-vertex data from these buffers and feeds this into the vertex shader as inputs.

Exercise 1.2 used a `VkBuffer` to download image data from the GPU to the CPU. Exercise 1.4 does the reverse. We specify data in a host-visible staging buffer, and copy this to the final GPU-only buffer.

The above process assumes a system with dedicated VRAM, of which the majority is not host-visible. A system where all memory is host-visible (e.g., many integrated GPUs) could skip the staging buffer. Nevertheless, the process outlined here works for all kinds of systems, even if it performs unnecessary work in some cases.

Vulkan Memory Allocator Exercise 1.2 allocated the buffer with "raw" Vulkan calls. In Exercise 1.4 (and going forward), we will instead use the *Vulkan Memory Allocator (VMA)*. VMA simplifies the process of allocating both buffers and images by reducing the necessary boilerplate code in many common cases. Additionally, instead of creating one `VkDeviceMemory` allocation per resource (as we did in Exerercise 1.2), VMA maintains a small number of "large" `VkDeviceMemory` allocations and sub-allocates memory from these to the individual Vulkan resources. This is especially useful on platforms such as Windows, where the ↗number of `VkDeviceMemory` allocations is limited.

In some cases, VMA might decide to create a *dedicated* allocation for a resource regardless. A dedicated allocation means that it is backed by a custom `VkDeviceMemory` allocation that is not shared with other resources. VMA lists when this occurs in ↗its documentation. One specific case is that Vulkan can internally indicate when a dedicated allocation should be used for a certain resource (see ↗`VK_KHR_dedicated_allocation`, which was promoted to core Vulkan in version 1.1).

VMA is ↗documented quite exhaustively. Study the documentation briefly - there are several short examples in the ↗*Quick Start* section. In order to use VMA, one first must create a `VmaAllocator` object. VMA follows the Vulkan API conventions, so the code's general structure should look somewhat familiar.

To simplify the process, `labutils` provides the `Allocator` class (`labutils/allocator.{hpp,cpp}`). It is fully implemented and ready to go. Study the code briefly. An instance of the `Allocator` is already created in `main()` in Exercise 1.4.

Buffer utilities In Exercise 1.2, we defined a temporary `Buffer` class. It holds two Vulkan resources: the main `VkBuffer` handle, and a `VkDeviceMemory` handle referring to the memory allocation that backed the buffer. With VMA, the `VkDeviceMemory` handle is replaced by a ↗`VmaAllocation` handle instead. Exercise 1.4 introduces a wrapper class similar to the one defined in Exercise 1.2, except that is defined in the `labutils` project this time. Find it in `vkbuffer.{hpp,cpp}`.

The header additionally declares a `create_buffer` helper function. Before proceeding, the function needs to be implemented (`vkbuffer.cpp`). It just wraps the ↗`vmaCreateBuffer` function defined by VMA:

```
VkBufferCreateInfo bufferInfo{};
bufferInfo.sType  = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size   = aSize;
bufferInfo.usage  = aBufferUsage;

VmaAllocationCreateInfo allocInfo{};
allocInfo.flags   = aMemoryFlags;
allocInfo.usage   = aMemoryUsage;

VkBuffer buffer = VK_NULL_HANDLE;
VmaAllocation allocation = VK_NULL_HANDLE;

if( auto const res = vmaCreateBuffer( aAllocator.allocator, &bufferInfo, &allocInfo, ▽
  ▷  &buffer, &allocation, nullptr ); VK_SUCCESS != res )
{
   throw Error( "Unable to allocate buffer.\n"
      "vmaCreateBuffer() returned %s", to_string(res).c_str()
   );
}

return Buffer( aAllocator.allocator, buffer, allocation );
```

You are already familiar with the `VkBufferCreateInfo` structure and its parameters (Exercise 1.2). VMA accepts a few additional parameters relating to the underlying allocation via the ↗`VmaAllocationCreateInfo` structure. Study the documentation. Two fields, the `usage` and `flags`, are of particular interest. In the current version, the primary options for the `usage` field of type ↗`VmaMemoryUsage` are:

`VMA_MEMORY_USAGE_AUTO` Option recommended by VMA. VMA selects the best memory type based on buffer type and flags.

`VMA_MEMORY_USAGE_AUTO_PREFER_DEVICE` VMA selects the memory type with a preference for device memory (memory with the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` set).

`VMA_MEMORY_USAGE_AUTO_PREFER_HOST` VMA selects the memory type with a preference for host memory.

The exercises frequently just use `VMA_MEMORY_USAGE_AUTO`, which is recommended by the authors of VMA. In fact, the `create_buffer` specifies this as the default value for the `VmaMemoryUsage` argument.

> Earlier versions of VMA used a different set of `VMA_MEMORY_USAGE_*` options, specifically `GPU_ONLY`, `CPU_ONLY`, `CPU_TO_GPU` and `GPU_TO_CPU`. However, these are now deprecated, though the VMA library will accept them still. (I found these now-deprecated constants more descriptive, but sometimes it is best to get on with the times.)

The `flags` field is of type `VmaMemoryCreateFlags` allows us to specify additional usage requirements. See linked documentation for all possible values. For now, the most important ones are:

`VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT` The option asks VMA to ensure that the memory is accessible from the CPU/host, meaning that it is allocated from a memory type with the `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` bit set. It does not imply `HOST_CACHED`, meaning that we have to be careful about how the memory is accessed on the host. Writing to it once by `std::memcpy()` is OK.

`VMA_ALLOCATION_CREATE_HOST_ACCESS_RANDOM_BIT` The option asks VMA to ensure that the memory is accessible from the CPU/host and that is *also* `HOST_CACHED`. We would use this flag if we intended to randomly read/write/modify the memory on the host.

If no extra requirements exist, it is, of course, possible to set the flags to zero. VMA maintains a page with a few recommended usage patterns.

Buffer barriers Previously, in Exercise 1.2, we benefited from synchronization provided by the subpass dependencies and a fence. The subpass dependencies in particular introduced a barrier that ensured that the rendered image was available to the subsequent copy of the image into the buffer.

The subpass dependencies define synchronization relating to the image attachments that are being rendered to in the render pass. To ensure that the copy from the host-visible staging buffer to the final GPU-only buffer finishes before we start reading data from the buffers during rendering, we must use a buffer barrier.

Buffer barriers, like all kinds of pipeline barriers, are issued with the `vkCmdPipelineBarrier` function. The function can issue multiple kinds of barriers in a single call. For now, we will focus on the buffer barriers, defined via the `VkBufferMemoryBarrier` structure.

> A good overview of Vulkan synchronization and the different types of barriers can be found in this blog post.

Additional buffer barriers are required later, when dealing with uniform buffers. Exercise 1.4 therefore introduces a helper function, `buffer_barrier`. Declare the `buffer_barrier` function in `labutils/vkutil.hpp`:

```
void buffer_barrier(
	VkCommandBuffer,
	VkBuffer,
	VkAccessFlags aSrcAccessMask,
	VkAccessFlags aDstAccessMask,
	VkPipelineStageFlags aSrcStageMask,
	VkPipelineStageFlags aDstStageMask,
	VkDeviceSize aSize = VK_WHOLE_SIZE,
	VkDeviceSize aOffset = 0,
	uint32_t aSrcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
	uint32_t aDstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED
);
```

Buffer barriers apply to a specific buffer, identified by the `VkBuffer` argument. By default, the barrier will apply to the whole buffer. The optional arguments `aSize` and `aOffset` can be used to narrow down the barrier to only a subset of the buffer's contents. The access flags describe how the buffer was accessed before the barrier and how it will be accessed after the barrier. The pipeline stages describe (as with the semaphore synchronization in Exercise 1.3) which pipeline stages of commands recorded before the barrier must have completed (`aSrcStageMask`) before the pipeline stages of subsequent commands can commence (`aDstStageMask`). The final optional arguments (`aSrcQueueFamilyIndex` and `aDstQueueFamilyIndex`) can be used to transfer ownership of a buffer from one queue family to another.

The implementation of the `buffer_barrier` helper looks as follows (`labutils/vkutil.cpp`):

```
void buffer_barrier( VkCommandBuffer aCmdBuff, VkBuffer aBuffer, VkAccessFlags aSrcAccessMask, VkAccessFlags aDstAccessMask, VkPipelineStageFlags aSrcStageMask, VkPipelineStageFlags aDstStageMask, VkDeviceSize aSize, VkDeviceSize aOffset, uint32_t aSrcQueueFamilyIndex, uint32_t aDstQueueFamilyIndex )
{
   VkBufferMemoryBarrier bbarrier{};
   bbarrier.sType  = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
   bbarrier.srcAccessMask        = aSrcAccessMask;
   bbarrier.dstAccessMask        = aDstAccessMask;
   bbarrier.buffer               = aBuffer;
   bbarrier.size                 = aSize;
   bbarrier.offset               = aOffset;
   bbarrier.srcQueueFamilyIndex  = aSrcQueueFamilyIndex;
   bbarrier.dstQueueFamilyIndex  = aDstQueueFamilyIndex;

   vkCmdPipelineBarrier(
      aCmdBuff,
      aSrcStageMask, aDstStageMask,
      0,
      0, nullptr,
      1, &bbarrier,
      0, nullptr
   );
}
```

As with previous helper functions, `buffer_barrier` focuses on simplicity. The underlying `vkCmdPipelineBarrier` can be used to define multiple barriers in a single call. This simplified helper function does not take advantage of this functionality.

Vertex buffer creation With this, we have the tools in place to create the vertex buffers. This exercises uses two separate buffers, one for vertex positions and one for per-vertex colors. The positions are identical to the ones used so far. As indicated earlier, Exercise 1.4 assumes a system with dedicated VRAM which is not fully host-visible. To create an on-GPU vertex buffer, the following steps are required:

1. Create on-GPU buffer
2. Create CPU/host-visible staging buffer
3. Place data into the staging buffer (`std::memcpy`)
4. Record commands to copy/transfer data from the staging buffer to the final on-GPU buffer
5. Record appropriate buffer barrier for the final on-GPU buffer
6. Submit commands for execution

For simplicity, Exercise 1.4 will use a separate function that performs the aforementioned steps. This function will create and destroy the necessary temporary resources. We must not destroy any resources while they are currently being used by pending/executing commands. A fence enables us to wait for the commands to finish before we destroy the resources.

Look at the `vertex_data.{hpp,cpp}` sources. The header defines a `ColorizedMesh` structure that holds the relevant buffers. Additionally, it defines a `create_triangle_mesh` function, which we will need to implement.

Find the function definition in `vertex_data.cpp`. The function already defines the relevant vertex-data data in a set of C/C++ arrays. Following the data definition, create the on-GPU buffers:

```
   // Create final position and color buffers
   lut::Buffer vertexPosGPU = lut::create_buffer(
      aAllocator,
      sizeof(positions),
      VK_BUFFER_USAGE_VERTEX_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
      0, // no additional VmaAllocationCreateFlags
      VMA_MEMORY_USAGE_AUTO_PREFER_DEVICE // or just VMA_MEMORY_USAGE_AUTO

   );
   lut::Buffer vertexColGPU = lut::create_buffer(
      aAllocator,
        sizeof(colors),
        VK_BUFFER_USAGE_VERTEX_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
        0, // no additional VmaAllocationCreateFlags
        VMA_MEMORY_USAGE_AUTO_PREFER_DEVICE // or just VMA_MEMORY_USAGE_AUTO
    );
```

This explicitly specifies `VMA_MEMORY_USAGE_AUTO_PREFER_DEVICE` to indicate that we want to use device local memory whenever possible.

Next, create the staging buffers. The `VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT` flag ensures that the memory is host visible. It is filled with `std::memcpy` later, so requesting sequential access is sufficient:

```
lut::Buffer posStaging = lut::create_buffer(
    aAllocator,
    sizeof(positions),
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT
);
lut::Buffer colStaging = lut::create_buffer(
    aAllocator,
    sizeof(colors),
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT
);
```

The staging buffers are CPU/host-visible. We can fill them by mapping the buffers to retrieve a normal C/C++ pointer to their contents, copying the data to this pointer and then unmapping the buffers again. We previously used `vkMapMemory`/`vkUnmapMemory` for this; with VMA, we must instead use VMA's `vmaMapMemory` and `vmaUnmapMemory`:

```
void* posPtr = nullptr;
if( auto const res = vmaMapMemory( aAllocator.allocator, posStaging.allocation, &posPtr ); VK_SUCCESS != res )
{
    throw lut::Error( "Mapping memory for writing\n"
        "vmaMapMemory() returned %s", lut::to_string(res).c_str()
    );
}
std::memcpy( posPtr, positions, sizeof(positions) );
vmaUnmapMemory( aAllocator.allocator, posStaging.allocation );

void* colPtr = nullptr;
if( auto const res = vmaMapMemory( aAllocator.allocator, colStaging.allocation, &colPtr ); VK_SUCCESS != res )
{
    throw lut::Error( "Mapping memory for writing\n"
        "vmaMapMemory() returned %s", lut::to_string(res).c_str()
    );
}
std::memcpy( colPtr, colors, sizeof(colors) );
vmaUnmapMemory( aAllocator.allocator, colStaging.allocation );
```

The next step is preparation for issuing the transfer commands that copy data from the staging buffers to the final on-GPU buffers. For this, we create a fence, a temporary command pool, and allocate a command buffer from the command pool with the utilities we have introduced in previous exercises:

```
// We need to ensure that the Vulkan resources are alive until all the transfers have completed. For simplicity,
// we will just wait for the operations to complete with a fence. A more complex solution might want to queue
// transfers, let these take place in the background while performing other tasks.
lut::Fence uploadComplete = create_fence( aContext );

// Queue data uploads from staging buffers to the final buffers.
// This uses a separate command pool for simplicity.
lut::CommandPool uploadPool = create_command_pool( aContext );
VkCommandBuffer uploadCmd = alloc_command_buffer( aContext, uploadPool.handle );
```

We then record the copy commands into the command buffer:

```
VkCommandBufferBeginInfo beginInfo{};
beginInfo.sType  = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
beginInfo.flags             = 0;
beginInfo.pInheritanceInfo  = nullptr;

if( auto const res = vkBeginCommandBuffer( uploadCmd, &beginInfo ); VK_SUCCESS != res )
{
   throw lut::Error( "Beginning command buffer recording\n"
      "vkBeginCommandBuffer() returned %s", lut::to_string(res).c_str()
   );
}

VkBufferCopy pcopy{};
pcopy.size  = sizeof(positions);

vkCmdCopyBuffer( uploadCmd, posStaging.buffer, vertexPosGPU.buffer, 1, &pcopy );

lut::buffer_barrier( uploadCmd,
   vertexPosGPU.buffer,
   VK_ACCESS_TRANSFER_WRITE_BIT,
   VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT,
   VK_PIPELINE_STAGE_TRANSFER_BIT,
   VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
);

VkBufferCopy ccopy{};
ccopy.size  = sizeof(colors);

vkCmdCopyBuffer( uploadCmd, colStaging.buffer, vertexColGPU.buffer, 1, &ccopy );

lut::buffer_barrier( uploadCmd,
   vertexColGPU.buffer,
   VK_ACCESS_TRANSFER_WRITE_BIT,
   VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT,
   VK_PIPELINE_STAGE_TRANSFER_BIT,
   VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
);

if( auto const res = vkEndCommandBuffer( uploadCmd ); VK_SUCCESS != res )
{
   throw lut::Error( "Ending command buffer recording\n"
      "vkEndCommandBuffer() returned %s", lut::to_string(res).c_str()
   );
}
```

The code uses two barriers, one for each of the final on-GPU buffers. The barriers declare that the results of the transfer operations (`VK_ACCESS_TRANSFER_WRITE_BIT` and `VK_PIPELINE_STAGE_TRANSFER_BIT`) must be visible to subsequent uses of the buffer when reading vertex attribute data from it (as indicated by the `VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT` access mask) ahead of vertex shader execution (i.e., before subsequent command's `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` pipeline stages execute).

You can find an overview of the Vulkan pipeline in the Vulkan 1.1 Quick Reference, on Page 5 (unfortunately, it does not seem that the quick reference has been updated for Vulkan 1.2 and Vulkan 1.3). You will find the *Input Assembler* stage ahead of the *Vertex Shader*. The input assembler (=`STAGE_VERTEX_INPUT`) is responsible for reading the per-vertex data from memory and preparing it for the vertex shader execution that follows.

Once recorded, we submit the command buffer for execution, and wait for the commands to finish:

```
// Submit transfer commands
VkSubmitInfo submitInfo{};
submitInfo.sType  = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount    = 1;
submitInfo.pCommandBuffers       = &uploadCmd;
```

```

if( auto const res = vkQueueSubmit( aContext.graphicsQueue, 1, &submitInfo, ▽
  ▷ uploadComplete.handle ); VK_SUCCESS != res )
{
   throw lut::Error( "Submitting commands\n"
      "vkQueueSubmit() returned %s", lut::to_string(res).c_str()
   );
}

// Wait for commands to finish before we destroy the temporary resources required for the transfers (staging
// buffers, command pool, ...)
//
// The code doesn't destory the resources implicitly – the resources are destroyed by the destructors of the
// labutils wrappers for the various objects once we leave the function's scope.
if( auto const res = vkWaitForFences( aContext.device, 1, &uploadComplete.handle, ▽
  ▷ VK_TRUE, std::numeric_limits<std::uint64_t>::max() ); VK_SUCCESS != res )
{
   throw lut::Error( "Waiting for upload to complete\n"
      "vkWaitForFences() returned %s", lut::to_string(res).c_str()
   );
}
```

Finally, we return the `ColorizedMesh` with the on-GPU buffers:

```
return ColorizedMesh{
   std::move(vertexPosGPU),
   std::move(vertexColGPU),
   sizeof(positions) / sizeof(float) / 2 // two floats per position
};
```

The presented implementation of `create_triangle_mesh` favours simplicity over performance. There are several improvements that one should consider in a non-tutorial setting (and especially for a more generalized version that is reused many times to upload vertex data). For instance, creating repeatedly creating and tearing down command pools is likely quite wasteful. Creating a single command pool and allocating multiple command buffers from it is a much better strategy. Further, the code uses a fence to wait for all transfers to complete, such that the temporary resources can be destroyed before returning from the method. Instead of waiting, the program could continue with other tasks while the transfers complete (e.g., loading and starting transfers for other data). This would require a bit smarter management of the temporary resources introduced by the code.

Graphics pipeline With the data in place, we must modify the graphics pipeline to accept input vertex attributes from the buffers. The buffers themselves are not typed, so we must also declare the format of the data that resides in the buffers. We also need to update the shaders themselves to accept and use the per-vertex inputs in the vertex shader.

Since we are moving to a more general graphics pipeline, we will rename a few things. First, instead of using the shaders `triangle.{vert,frag}`, we will switch to a pair of shaders named `shader2d.vert` and `shader2d.frag`. Update the `kVertShaderPath` and `kFragShaderPath` constants in `main.cpp` appropriately. (Don't forget to re-run Premake to generate updated project/make-files!)

In the vertex shader we will now get a vertex position and color as input. The position we simply write to the `gl_Position` built-in (after expanding to a `vec4`). The color we output for the fragment shading stage. The colors from the different vertices will be interpolated during rasterization to create the appropriate per-fragment inputs. Hence, the `exercise4/shaders/shader2d.vert` shader becomes:

```
#version 450

layout( location = 0 ) in vec2 iPosition;
layout( location = 1 ) in vec3 iColor;

layout( location = 0 ) out vec3 v2fColor; // v2f = "vertex to fragment"

void main()
{
   v2fColor = iColor;
```

```
    gl_Position = vec4( iPosition, 0.5f, 1.f );
}
```

Note the `location` indices on the `in` inputs. These will be used to link vertex inputs defined in Vulkan to the GLSL shaders. Specifically, when we configure the graphics pipeline, the indices in the shaders must match the indices that we specify in the `VkPipelineVertexInputStateCreateInfo`. The `location` on the vertex shader's `out` output must match the corresponding `in` input of the fragment shader. Implement the fragment shader (`exercise4/shaders/shader2d.frag`) as well:

```
#version 450

layout( location = 0 ) in vec3 v2fColor;

layout( location = 0 ) out vec4 oColor;

void main()
{
    oColor = vec4( v2fColor, 1.f );
}
```

The fragment shader simply takes the interpolated color input and copies it to the output (which will be written to the framebuffer's color attachment with index 0).

Next, rename the `create_triangle_pipeline_layout` function to just `create_pipeline_layout`. It will remain unchanged compared to Exercise 1.3 otherwise (transfer your solution from there). Do the same for the `create_triangle_pipeline` (to `create_pipeline`). We will need to modify the latter however.

Exercise 1.3 did not use vertex inputs, and consequently left the `VkPipelineVertexInputStateCreateInfo` structure "empty". To pull data from the vertex buffers, we will need to configure the graphics pipeline to do so using the ↗`VkPipelineVertexInputStateCreateInfo` structure. It accepts two sets of structures:

- ↗`VkVertexInputBindingDescription` and
- ↗`VkVertexInputAttributeDescription`.

The former (`VkVertexInputBindingDescription`) declares how data is read from buffers, mainly indicating where the data for each vertex can be found. The latter (`VkVertexInputAttributeDescription`) defines how the data is mapped to the vertex shader's inputs.

We are reading from two separate buffers, therefore use two `VkVertexInputBindingDescription` instances:

```
VkVertexInputBindingDescription vertexInputs[2]{};
vertexInputs[0].binding    = 0;
vertexInputs[0].stride     = sizeof(float)*2;
vertexInputs[0].inputRate  = VK_VERTEX_INPUT_RATE_VERTEX;

vertexInputs[1].binding    = 1;
vertexInputs[1].stride     = sizeof(float)*3;
vertexInputs[1].inputRate  = VK_VERTEX_INPUT_RATE_VERTEX;
```

The `binding` member identifies the index at which we bind the corresponding input buffer when drawing (with `vkCmdBindVertexBuffers`, introduced later). We will bind the position buffer at index zero and the color buffer at index one. The `stride` identifies the number of bytes between two consecutive elements in the buffer. Positions consist of two floats, but the colors use three floats. There is no additional padding between consecutive vertices. Finally, `inputRate` member identifies how frequently a new input should be fetched from the buffer. The most common option is `VK_VERTEX_INPUT_RATE_VERTEX`, meaning that each vertex gets a new value from the buffer. The only other option is `VK_VEREX_INPUT_RATE_INSTANCE`, which is useful for instanced drawing (all vertices in an instance share the same value).

The vertex shader expects two inputs, the position and the color. Consequently, these are described with two `VkVertexInputAttributeDescription` instances:

```
    VkVertexInputAttributeDescription vertexAttributes[2]{};
    vertexAttributes[0].binding   = 0; // must match binding above
    vertexAttributes[0].location  = 0; // must match shader
    vertexAttributes[0].format    = VK_FORMAT_R32G32_SFLOAT;
    vertexAttributes[0].offset    = 0;

    vertexAttributes[1].binding   = 1; // must match binding above
    vertexAttributes[1].location  = 1; // must match shader
    vertexAttributes[1].format    = VK_FORMAT_R32G32B32_SFLOAT;
    vertexAttributes[1].offset    = 0;
```

The `binding` member identifies which of the buffer bindings (first set of structures) the inputs stem from. The `location` member defines which input attribute in the shader the data is sent to – this is the index that must match the corresponding `location` declaration in the veretx shader. The `format` member defines the format of the data. Finally, the `offset` may be used if multiple attributes are packed into a single buffer/binding. This is not the case here, and in each case, the relevant data is at offset zero of the binding.

The settings are passed to `vkCreateGraphicsPipeline` via the `VkPipelineVertexInputStateCreateInfo` structure:

```
  inputInfo.vertexBindingDescriptionCount    = 2; // number of vertexInputs above
  inputInfo.pVertexBindingDescriptions       = vertexInputs;
  inputInfo.vertexAttributeDescriptionCount  = 2; // number of vertexAttributes above
  inputInfo.pVertexAttributeDescriptions     = vertexAttributes;
```

Drawing When submitting the drawing command (`vkCmdDraw`), we must specify the buffers from which the draw command sources its vertex input data. This is done with the `vkCmdBindVertexBuffer`. (Previously, the graphics pipeline did not declare any vertex input data, so we were able to skip this step.)

With this, one way of drawing multiple objects also becomes apparent. We simply store each object's vertex data in a separate set of `VkBuffer`s. To render, we simply loop over the objects, and for each object, bind the object's buffers with `vkCmdBindVertexBuffers` and issue a `vkCmdDraw`.
This is indeed a simple way of organizing the data when needing to handle multiple objects. More complex patterns may want to avoid having to call `vkCmdBindVertexBuffers` for each object, and instead store the data for multiple objects in a single set of `VkBuffer`s. Objects can then be drawn individually by using the `firstVertex` argument of `vkCmdDraw` to specify where each object's data is located in the merged buffers. One can then even avoid issuing a separate `vkCmdDraw` per object, via functions such as `vkCmdDrawIndirect`. However, for simplicity, the exercises will stick to the simple pattern with per-object buffers and per-object `vkCmdDraw` calls.

Draw commands are issued in `record_commands`. Transfer your solution from Exercise 3. We will then need to pass a few additional arguments to `record_commands`:

- `VkBuffer aPositionBuffer`
- `VkBuffer aColorBuffer`
- `std::uint32_t aVertexCount`

Add these arguments to the declaration and definition of your `record_commands`. Update the code to pass the relevant values (e.g. both the `triangleMesh.positions.buffer`, `triangleMesh.colors.buffer` buffers and the `triangleMesh.vertexCount` count) when calling `record_commands` in `main()`.

In `record_commands`, bind the vertex buffers and update the existing call to `vkCmdDraw` as follows:

```
  // Bind vertex input
  VkBuffer buffers[2] = { aPositionBuffer, aColorBuffer };
  VkDeviceSize offsets[2]{};

  vkCmdBindVertexBuffers( aCmdBuff, 0, 2, buffers, offsets );

  // Draw vertices
  vkCmdDraw( aCmdBuff, aVertexCount, 1, 0, 0 );
```

There are likely a few other functions that are not yet implemented in `main.cpp`. Transfer the relevant solutions from Exercise 1.3 for these. At that point, you should be able to successfully run the program, and see a result similar the one shown in Figure 2.

You may want to experiment with adding a few additional vertices (e.g., a second and third triangle). Note that you only need to change the data defined in the `vertex_data.cpp` file – we no longer hardcode anything in the main program or in the shaders.

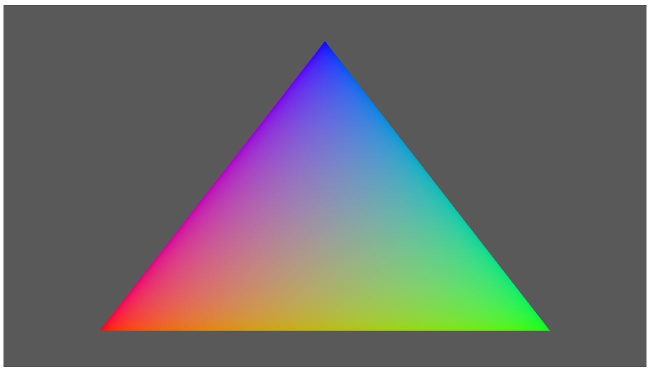


Figure 2: *Triangle drawn with vertex buffers (Section 2).*

3 Uniform Buffer & Descriptors

(Uniform Block (GLSL) — Desc. Set Layout — Pipeline Layout — Uniform Buffer — Desc. Set Pool — Desc. Set — Drawing)

At this point, we have been staring at a single 2D triangle for way too long. It is about time to move to 3D. First, we need to extend the input data from specifying 2D positions to specifying 3D positions. We then need to define a projection matrix and pass it to the vertex shader. This is done with an uniform interface block (GLSL) and a uniform buffer (Vulkan).

The setup in GLSL is fairly simply. We just need to declare the corresponding uniform interface block. The Vulkan side requires a few more steps. First, we need to define a descriptor set layout. A descriptor set is a group of associated uniform resources. The set layout describes what types of resources the set contains – think of this like a C/C++ struct declaration. Next, we need specify which types of descriptor sets the pipeline uses – if we imagine the pipeline to be a C/C++ function, this would roughly correspond to declaring the arguments that the function accepts (in addition to the vertex inputs that we already declared; the analogy is not perfect). This is done through the pipeline layout. Next, we will create the uniform buffer instance (a `VkBuffer`). This is where we store the actual matrix that is passed to the shader. Finally, we will create a descriptor set, and initialize it such that it refers to the created buffer – think of this as creating an instance of the previously declared struct and initializing it with the correct values.

Input data We start with extending the input vertex data to provide 3D positions. Define a new function `create_plane_mesh` in `vertex_data.hpp`:

```
ColorizedMesh create_plane_mesh( labutils::VulkanContext const&, labutils::Allocator ▽
    ▷ const& );
```

We can reuse the `ColorizedMesh` for now.

Define the function in `vertex_data.cpp`. As the name of the function indicates, it will create a mesh that represents a plane, in this case aligned with the XY plane. The plane is made from two triangles:

```
// Vertex data
static float const positions[] = {
    -1.f, 0.f, -6.f, // v0
    -1.f, 0.f, +6.f, // v1
    +1.f, 0.f, +6.f, // v2

    -1.f, 0.f, -6.f, // v0
    +1.f, 0.f, +6.f, // v2
    +1.f, 0.f, -6.f  // v3
};
static float const colors[] = {
    0.4f, 0.4f, 1.0f, // c0
    0.4f, 1.0f, 0.4f, // c1
    1.0f, 0.4f, 0.4f, // c2

    0.4f, 0.4f, 1.0f, // c0
    1.0f, 0.4f, 0.4f, // c2
    1.0f, 0.4f, 0.0f  // c3
};
```

Implement the function otherwise as described in Section 2. *Do not forget to change the return statement to indicate the correct number of vertices:*

```
return ColorizedMesh{
   std::move(vertexPosGPU),
   std::move(vertexColGPU),
   sizeof(positions) / sizeof(float) / 3 // now three floats per position
};
```

Change the call to `create_triangle_mesh` to the newly created `create_plane_mesh`:

```
//ColorizedMesh triangleMesh = create_triangle_mesh( window, allocator ); // old
ColorizedMesh planeMesh = create_plane_mesh( window, allocator );
```

Also update the references to `triangleMesh` to `planeMesh` in the rest of the code.

We additionally need to update the vertex input information when creating the graphics pipeline:

```
VkVertexInputBindingDescription vertexInputs[2]{};
vertexInputs[0].binding    = 0;
vertexInputs[0].stride     = sizeof(float)*3; // Changed!
vertexInputs[0].inputRate  = VK_VERTEX_INPUT_RATE_VERTEX;

// ...

VkVertexInputAttributeDescription vertexAttributes[2]{};
vertexAttributes[0].binding   = 0; // must match above
vertexAttributes[0].location  = 0; // must match shader
vertexAttributes[0].format    = VK_FORMAT_R32G32B32_SFLOAT; // Changed!
vertexAttributes[0].offset    = 0;

// ...
```

I.e., each position now consists of three floats, rather than two as before.

GLSL uniform block The 3D position input that the vertex shader receives needs to be transformed with the model-view-projection matrix. Exercise 1.4 simplifies this slightly, by assuming that the vertex positions are defined in world space, meaning that we do not need a separate model transform. Hence, to compute the clip space position $\mathbf{p}'$, we transform an input vertex position $\mathbf{p}$ with the camera matrix C and then with the projection matrix P:

$$\mathbf{p}' = P\,C\,\mathbf{p}$$

Matrix transformations are associative, so we can choose the order in which they are computed. It is particularly useful to first multiply the two matrices P and C, and then transform the vertex position with the resulting matrix:

$$\mathbf{p}' = (P\,C)\;\mathbf{p}$$

Written this way, it is obvious that the matrix product $P\,C$ does not depend on the vertex position, and can therefore be precomputed outside of the vertex shader (on the host, once per frame). With $Q = P\,C$, the transformation in the vertex shader becomes a single matrix multiplication

$$\mathbf{p}' = Q\,\mathbf{p}.$$

In terms of code, this means that the GLSL shader needs to receive the matrix Q somehow. We do so through an uniform interface block:

```
layout( set = 0, binding = 0 ) uniform UScene
{
   mat4 camera;
   mat4 projection;
   mat4 projCam;
} uScene;
```

For completeness, we also pass in the C (`camera`) and P (`projection`) matrices individually (although they will remain unused for now).

The input to the vertex shader is (as per previous section) no longer a `vec2`, but rather a `vec3`:

```
layout( location = 0 ) in vec3 iPosition; // note: vec3 instead of vec2
```

Finally, we need to transform the input position with the provided matrix:

```
gl_Position = uScene.projCam * vec4( iPosition, 1.f );
```

Implement this in the `shader3d.vert` and `shader3d.frag` shader pair in the `shaders` subdirectory. The `shader3d.frag` is unchanged from the `shader2d.frag` presented in Section 2, so copy it over. The complete `shader3d.vert` vertex shader will look as follows:

```
#version 450

layout( location = 0 ) in vec3 iPosition;
layout( location = 1 ) in vec3 iColor;

layout( set = 0, binding = 0 ) uniform UScene
{
   mat4 camera;
   mat4 projection;
   mat4 projCam;
} uScene;

layout( location = 0 ) out vec3 v2fColor;

void main()
{
   v2fColor = iColor;

   gl_Position = uScene.projCam * vec4( iPosition, 1.f );
}
```

Make sure to update the `kVertShaderPath` and `kFragShaderPath` constants in `main.cpp` to refer to this new pair of shaders.

Descriptor set layout On the Vulkan side, we start with defining a descriptor set layout. Uniform resources are grouped into sets. At the moment, the shaders use a single uniform resource, a uniform block, in a single uniform set. This uniform block is the first member of the set, as indicated by the `binding = 0` declaration in the GLSL code above. Currently no other uniform resources are being used in any of the shaders referenced by the graphics pipeline.

We need to create a descriptor set layout that mirrors this information on the Vulkan side. Locate the definition of `create_scene_descriptor_layout`. Descriptor set layouts are created with

- `vkCreateDescriptorSetLayout` and
- `VkDescriptorSetLayoutCreateInfo`.

Looking at the definition of `VkDescriptorSetLayoutCreateInfo`, it is mainly a list of one or more instances of `VkDescriptorSetLayoutBinding`. Each `VkDescriptorSetLayoutBinding` corresponds to one member of the descriptor set. As there is a single member in the descriptor set, a uniform block/buffer, the list of `VkDecriptorSetLayoutBinding` becomes:

```
   VkDescriptorSetLayoutBinding bindings[1]{};
   bindings[0].binding          = 0; // number must match the index of the corresponding
                                     // binding = N declaration in the shader(s)!
   bindings[0].descriptorType   = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
   bindings[0].descriptorCount  = 1;
   bindings[0].stageFlags       = VK_SHADER_STAGE_VERTEX_BIT;
```

Here, we specify the binding, which must match the one we specified earlier in the shader (i.e. zero). We then declare the type of descriptor entry the binding holds, an uniform buffer in this case. This must also match the type in the shader (e.g., a uniform interface block becomes an uniform buffer). Finally, we define the shader stages that will use this descriptor. In this case, we only access the matrices in the vertex shader stage.

Next, we create the descriptor set layout (`VkDescriptorSetLayout`):

```
   VkDescriptorSetLayoutCreateInfo layoutInfo{};
   layoutInfo.sType         = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
```

```
layoutInfo.bindingCount  = sizeof(bindings)/sizeof(bindings[0]);
layoutInfo.pBindings     = bindings;

VkDescriptorSetLayout layout = VK_NULL_HANDLE;
if( auto const res = vkCreateDescriptorSetLayout( aWindow.device, &layoutInfo, ▽
  ▷ nullptr, &layout ); VK_SUCCESS != res )
{
   throw lut::Error( "Unable to create descriptor set layout\n"
      "vkCreateDescriptorSetLayout() returned %s", lut::to_string(res).c_str()
   );
}

return lut::DescriptorSetLayout( aWindow.device, layout );
```

Finally, we need to call `create_scene_descriptor_layout`. In `main()`, add a call to the function, for example, just before creating the pipeline layout (`create_pipeline_layout`):

```
lut::DescriptorSetLayout sceneLayout = create_scene_descriptor_layout( window );
```

Pipeline layout Each type of descriptor set, as identified by a descriptor set layout, that a pipeline uses needs to be specified when creating a pipeline. This is done with a pipeline layout. Exercise 1.2 first introduced the pipeline layout object (`VkPipelineLayout`), but at that point it was left "empty" as the pipeline did not have any uniform inputs.

Find the declaration of `create_pipeline_layout`. Add an argument of type `VkDescriptorSetLayout`. Also update the code to pass the `VkDescriptorSetLayout` (i.e., `sceneLayout.handle`) handle that we just created to it to the function when it is called.

The implementation of `create_pipeline_layout` needs to pass the list of descriptor set layouts that will be used by the pipeline to the `vkCreatePipelineLayout` function via the `pSetLayouts` and `setLayoutCount` members of the `VkPipelineLayoutCreateInfo` structure:

```
VkDescriptorSetLayout layouts[] = {
   // Order must match the set = N in the shaders
   aSceneLayout  // set 0
};

VkPipelineLayoutCreateInfo layoutInfo{};
layoutInfo.sType  = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
layoutInfo.setLayoutCount          = sizeof(layouts)/sizeof(layouts[0]); // updated!
layoutInfo.pSetLayouts             = layouts; // updated!
layoutInfo.pushConstantRangeCount  = 0;
layoutInfo.pPushConstantRanges     = nullptr;

VkPipelineLayout layout = VK_NULL_HANDLE;
if( auto const res = vkCreatePipelineLayout( aContext.device, &layoutInfo, nullptr, ▽
  ▷ &layout ); VK_SUCCESS != res )
{
   throw lut::Error( "Unable to create pipeline layout\n"
      "vkCreatePipelineLayout() returned %s", lut::to_string(res).c_str()
   );
}

return lut::PipelineLayout( aContext.device, layout );
```

The descriptor set layouts must be provided in the right order. The first element of the array pointed to by `pSetLayouts` corresponds to the set with index zero (i.e., the collection of uniform inputs that have `set = 0` in their declaration in GLSL). Although there is none at the moment, a second element in the array would correspond to to the uniform inputs with `set = 1` in the GLSL shaders.

While we are free to choose how we distribute the various uniform inputs (uniform buffers, textures, shader storage buffers, and so on) between different sets, it typically makes sense to group uniform inputs by how often they change.
For example, all uniform inputs that change once per frame might go into set zero, all uniform inputs that change once per "object" might go into set one, and so on. You will see later that it is easy to change a complete descriptor set in one go.
This exercise takes this approach. The descriptor set layout that we create is called (by default) `sceneLayout` to indicate that it is a descriptor related to per-scene uniform data.

Vulkan uniform buffer With the pipeline layout, we have declared, via the descriptor set layout, that the pipeline expects a uniform buffer as an input. The next step is to create this buffer, along with the infrastructure required to provide the data to it in our CPU program.

Recall (Uniform Block (GLSL)) that the uniform interface block expects three `mat4` 4×4 matrices. Exercise 1.4 will define a C/C++ structure that mirrors this.

Look for the declaration of `SceneUniform` in `main.cpp`. We will need to first define the structure:

```
struct SceneUniform
{
    // Note: need to be careful about the packing/alignment here!
    glm::mat4 camera;
    glm::mat4 projection;
    glm::mat4 projCam;
};

// We want to use vkCmdUpdateBuffer() to update the contents of our uniform buffers. vkCmdUpdateBuffer()
// has a number of requirements, including the two below. See
// https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/vkCmdUpdateBuffer.html
static_assert( sizeof(SceneUniform) <= 65536, "SceneUniform must be less than 65536 bytes for vkCmdUpdateBuffer" );
static_assert( sizeof(SceneUniform) % 4 == 0, "SceneUniform size must be a multiple of 4 bytes" );
```

We must take care that the C/C++ structure matches exactly the memory layout that the shaders expect. By default, uniform interface blocks are packed according to the *std140* layout. The layout is documented in Section 7.6.2.2 ("Standard Uniform Block Layout") in the OpenGL 4.5 specification (link). Since we only have three 4×4 `float` matrices, we are pretty safe.

As indicated by the comments in the above code, Exercise 1.4 plans to use the `vkCmdUpdateBuffer` function to update the contents of the uniform buffer based on this structure. The `vkCmdUpdateBuffer` function has a few requirements, including that the data passed to it is at most 65536 bytes ($= 64$k), and that the size of the data is a multiple of 4. The `static_assert` lines check this at compile time – if the structure violates these requirements, it will cause a compile-time error (and you need to fix the structure).

Whenever we change the uniform block interface in the shader(s), we must update the C/C++ definition of this structure to match as well (and vice-versa). To indicate that this structure needs special consideration, it is placed in an explicit `glsl` namespace. You are not required to follow this convention in your own code; however, if you do not, I would recommend coming up with something similar.

Next, we create a `VkBuffer` to hold the uniform data in GPU memory. Fortunately, this is quite simple, thanks to the `labutils::create_buffer` method that we defined earlier.

Create a buffer in `main()`, e.g., after creating the vertex input data (after the call to `create_plane_mesh`):

```
lut::Buffer sceneUBO = lut::create_buffer(
    allocator,
    sizeof(glsl::SceneUniform),
    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
    0,
    VMA_MEMORY_USAGE_AUTO_PREFER_DEVICE
);
```

The usage flags contain the new `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT`, with which we indicate to Vulkan that this buffer is intended to be used as an uniform buffer. Additionally, as we intend to copy data into the buffer, we need to specify the `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag.

Uniform buffers are one of the resources that could easily be placed in different memory types. It might be possible to keep UBOs in non-device-local host-visible memory, assuming that the contents would end up being cached on the GPU. Unfortunately, there is no simple answer. As mentioned here, if there were only one right answer, Vulkan would not let its users pick.

Descriptor set pool The next step is to create a descriptor set (`VkDescriptorSet`) of the descriptor set type defined by the layout created earlier (Desc. Set Layout) and initialize it to have its binding zero refer to the uniform buffer that we just created.

Descriptor sets, specifically the individual descriptor resources, are allocated from a pool (`VkDescriptorPool`). To allocate a descriptor set, we first need to create the pool.

For Exercise 1.4, we will create a generic pool, and as such, we define the functions to create such in the shared `labutils` code. Find the `labutils/vkutil.hpp` header and add the following two function declarations:

```
DescriptorPool create_descriptor_pool( VulkanContext const&, std::uint32_t ▽
  ▷ aMaxDescriptors = 2048, std::uint32_t aMaxSets = 1024 );
VkDescriptorSet alloc_desc_set( VulkanContext const&, VkDescriptorPool, ▽
  ▷ VkDescriptorSetLayout );
```

The two functions follow the same pattern that we already saw for command buffers and the command pool in Exercise 1.2. The relevant Vulkan functions are:

- `vkCreateDescriptorPool`,
- `VkDescriptorPoolCreateInfo`,
- `vkAllocateDescriptorSets` and
- `VkDescriptorSetAllocateInfo`.

The `create_descriptor_pool` function is implemented as follows (`labutils/vkutil.cpp`):

```
DescriptorPool create_descriptor_pool( VulkanContext const& aContext, std::uint32_t ▽
    ▷ aMaxDescriptors, std::uint32_t aMaxSets)
{
   VkDescriptorPoolSize const pools[] = {
      { VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, aMaxDescriptors },
      { VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, aMaxDescriptors }
   };

   VkDescriptorPoolCreateInfo poolInfo{};
   poolInfo.sType           = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
   poolInfo.maxSets         = aMaxSets;
   poolInfo.poolSizeCount   = sizeof(pools)/sizeof(pools[0]);
   poolInfo.pPoolSizes      = pools;

   VkDescriptorPool pool = VK_NULL_HANDLE;
   if( auto const res = vkCreateDescriptorPool( aContext.device, &poolInfo, nullptr, &▽
     ▷ pool ); VK_SUCCESS != res )
   {
      throw Error( "Unable to create descriptor pool\n"
         "vkCreateDescriptorPool() returned %s", to_string(res).c_str()
      );
   }

   return DescriptorPool( aContext.device, pool );
}
```

We first list the kinds of descriptors that the pool should contain. For now, these are

- `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`: descriptors referring to uniform buffers
- `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`: descriptors referring to textures (`sampler2D` in GLSL). The descriptors combining a `VkImage` and `VkSampler`. Descriptors of this type will be used in Section 4.

For each kind of descriptor that we intend to use, we need to define how many descriptors of that kind the pool should contain. For example, if we were to create a pool with two `UNIFORM_BUFFER` descriptors, we could allocate two descriptor sets with one uniform buffer descriptor each from the pool. We allocate a large number of descriptors of both types, hoping that it will be sufficient for our needs.

We must also specify the maximum number of descriptor sets that can be allocated from the pool (`maxSets` member). Allocating more descriptor sets than this from the pool will fail even if the individual descriptor resources are not exhausted yet. We deal with this in the same way as with the individual resources, and just specify a large number, and hope that we will not run out of descriptor sets.

Next, we define the `alloc_desc_set` function. This function allocates a descriptor set of a specific type (identified via the `VkDescriptorSetLayout` handle) from the pool:

```
VkDescriptorSet alloc_desc_set( VulkanContext const& aContext, VkDescriptorPool aPool, ↘
    ▷ VkDescriptorSetLayout aSetLayout )
{
   VkDescriptorSetAllocateInfo allocInfo{};
   allocInfo.sType  = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
   allocInfo.descriptorPool      = aPool;
   allocInfo.descriptorSetCount  = 1;
   allocInfo.pSetLayouts         = &aSetLayout;

   VkDescriptorSet dset = VK_NULL_HANDLE;
   if( auto const res = vkAllocateDescriptorSets( aContext.device, &allocInfo, &dset );↘
    ▷ VK_SUCCESS != res )
   {
      throw Error( "Unable to allocate descriptor set\n"
         "vkAllocateDescriptorSets() returned %s", to_string(res).c_str()
      );
   }

   return dset;
}
```

You might have noticed that the Vulkan function to allocate a descriptor set from a descriptor pool is called `vkAllocateDescriptorSets`, with the plural "s". As earlier, this indicates that Vulkan expects us to create many descriptor sets, and that the approach of allocating them one-by-one with the `alloc_desc_set` function may not be the ideal patter. However, for Exercise 1.4 it will suffice (we will only deal with a very small number of descriptor sets).

Finally, we will need to create a descriptor pool. In `main()`, add a call to `create_descriptor_pool`:

```
   lut::DescriptorPool dpool = lut::create_descriptor_pool( window );
```

Descriptor set With the tools in place, we can finally allocate the descriptor set and initialize it such that it refers to the previously created uniform buffer (Uniform Buffer).

We already defined the utility function to allocate a descriptor set out of a descriptor pool in the last section (`alloc_desc_set`). There are multiple ways one can initialize a descriptor set. Exercise 1.4 uses the basic `vkUpdateDescriptorSets` function. It accepts two lists that define what descriptor updates should take place. We will focus in the first one, consisting of one or more `VkWriteDescriptorSet` structures. (The second list enables copying data between descriptors, which Exercise 1.4 does not require.)

Each `VkWriteDescriptorSet` element defines how one descriptor in a specific descriptor set is written. As the naming of the function indicates, multiple descriptors from multiple different descriptor sets can be updated in a single call to `vkWriteDescriptorSets`.

The options that `VkWriteDescriptorSet` accepts are:

`dstSet` The `VkDescriptorSet` instance of which the specified descriptor (see `dstBinding` member below) should be updated.

`dstBinding` Index of the binding that should be updated.

`descriptorType` Type of descriptor. This must match the type for the binding that was specified when declaring the corresponding descriptor set layout.

pImageInfo Specifies the `VkImage` and/or `VkSampler` for descriptor types that refer to images and/or samplers. Will be used later for descriptors of type `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`.

pBufferInfo Specifies the `VkBuffer` for descriptor types that refer to buffers. We will use this for our descriptor of type `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`.

There are a few other members, but these are not used in Exercise 1.4. Refer to the Vulkan documentation for more information.

Hence, to initialize the descriptor at Binding 0 to the uniform buffer that we previously created, we need to create and fill a `VkDescriptorBufferInfo` with the relevant info. (The `range` member can be set to the special value `VK_WHOLE_SIZE` if the specified buffer only holds the data for the single uniform interface block.)

Putting everything together will result in the following code (in `main()`):

```
VkDescriptorSet sceneDescriptors = lut::alloc_desc_set( window, dpool.handle, ▽
  ▷ sceneLayout.handle );

{
   VkWriteDescriptorSet desc[1]{};

   VkDescriptorBufferInfo sceneUboInfo{};
   sceneUboInfo.buffer  = sceneUBO.buffer;
   sceneUboInfo.range   = VK_WHOLE_SIZE;

   desc[0].sType   = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
   desc[0].dstSet           = sceneDescriptors;
   desc[0].dstBinding       = 0;
   desc[0].descriptorType   = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
   desc[0].descriptorCount  = 1;
   desc[0].pBufferInfo      = &sceneUboInfo;


   constexpr auto numSets = sizeof(desc)/sizeof(desc[0]);
   vkUpdateDescriptorSets( window.device, numSets, desc, 0, nullptr );
}
```

Drawing With all the resources in place, we finally get to drawing.

First, we need to update the uniform buffer object with the relevant data. For simplicity, the necessary values (matrices) will simply be computed each frame. Implement the `update_scene_uniforms` method to compute the matrices:

```
float const aspect = aFramebufferWidth / float(aFramebufferHeight);

aSceneUniforms.projection = glm::perspectiveRH_ZO(
   lut::Radians( cfg::kCameraFov ).value(),
   aspect,
   cfg::kCameraNear,
   cfg::kCameraFar
);
aSceneUniforms.projection[1][1] *= -1.f; // mirror Y axis

aSceneUniforms.camera = glm::translate( glm::vec3( 0.f, -0.3f, -1.f ) );

aSceneUniforms.projCam = aSceneUniforms.projection * aSceneUniforms.camera;
```

The projection matrix is created with GLM's `glm::perspectiveRH_ZO` function. The `RH` indicates a right handed clip space, and the `ZO` indicates that the clip space extends from zero to one along the Z-axis.

Important: Vulkan's clip space differs from the OpenGL clip space. OpenGL's clip space extends from -1 to 1 on all axes (after homogenization). Vulkan's clip space instead extends from -1 to 1 along the X and Y axes, but 0 to 1 in the Z direction. Using the wrong projection matrix will likely lead to some rather strange behaviour.

The plane that we previously created exists at $Y = 0$, and is centered around the origin. To make sure it is easily visible, the code places the camera slightly up and a bit back.

The code then precomputes the `projCam` matrix that combines the projection and camera transformations into a single matrix. This happens once per frame. But, as a consequence, each vertex is transformed with a single matrix multiplication. Transforming the vertex position with both matrices (or worse, performing the matrix multiplication in the shader, once for each vertex) would be quite a bit more expensive.

Add the call to `update_scene_uniforms` inside of the main loop:

```
// Prepare data for this frame
glsl::SceneUniform sceneUniforms{};
update_scene_uniforms( sceneUniforms, window.swapchainExtent.width, window.▽
  ▷ swapchainExtent.height );
```

We then need to copy the contents of the `sceneUniforms` instance to the `VkBuffer`. As mentioned earlier, Exercise 1.4 proposes to use ↗`vkCmdUpdateBuffer` to perform the copy. The `vkCmd` prefix on the function indicates that this is a command that needs to be recorded into a command buffer. We will therefore add this command to the commands recorded in `record_commands`.

The `record_commands` method will require a few additional arguments:

- A `VkBuffer` argument to pass in the handle to the uniform buffer.
- A (`const`) reference to the `SceneUniform` structure
- The `VkPipelineLayout` handle, which will be required to bind the descriptor set ahead of the draw call
- The `VkDescriptorSet` handle to the descriptor set that references the uniform buffer

At this point, the `record_commands` declaration might look like the following:

```
void record_commands(
   VkCommandBuffer,
   VkRenderPass,
   VkFramebuffer,
   VkPipeline,
   VkExtent2D const&,
   VkBuffer aPositionBuffer,
   VkBuffer aColorBuffer,
   std::uint32_t aVertexCount,
   VkBuffer aSceneUBO,
   glsl::SceneUniform const&,
   VkPipelineLayout,
   VkDescriptorSet aSceneDescriptors
);
```

Make sure that all the necessary arguments are passed to `record_commands` in the main loop:

```
record_commands(
   cbuffers[imageIndex],
   renderPass.handle,
   framebuffers[imageIndex].handle,
   pipe.handle,
   window.swapchainExtent,
   planeMesh.positions.buffer,
   planeMesh.colors.buffer,
   planeMesh.vertexCount,
   sceneUBO.buffer,
   sceneUniforms,
   pipeLayout.handle,
   sceneDescriptors
);
```

We then implement the changes to the `record_commands` method. Adjust the definition to include the new arguments:

```
void record_commands( VkCommandBuffer aCmdBuff, VkRenderPass aRenderPass, ▽
  ▷ VkFramebuffer aFramebuffer, VkPipeline aGraphicsPipe, VkExtent2D const& ▽
  ▷ aImageExtent, VkBuffer aPositionBuffer, VkBuffer aColorBuffer, std::uint32_t ▽
  ▷ aVertexCount, VkBuffer aSceneUBO, glsl::SceneUniform const& aSceneUniform, ▽
  ▷ VkPipelineLayout aGraphicsLayout, VkDescriptorSet aSceneDescriptors )
```

Then, the first step is to update the uniform buffer with `vkCmdUpdateBuffer`. Do so immediately after starting command buffer recording, but before starting the render pass:

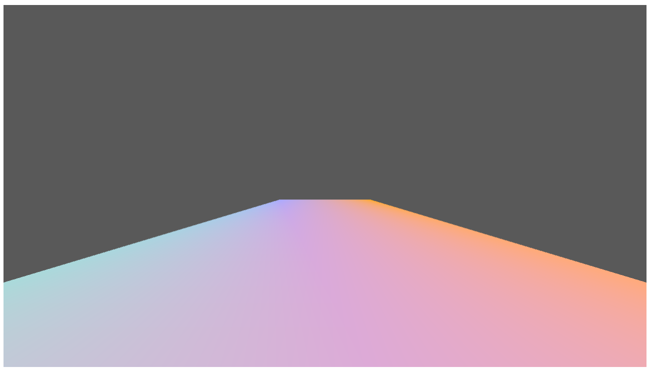


Figure 3: *A 3D plane (two triangles) drawn with a perspective projection (Section 3).*

```
// Upload scene uniforms
lut::buffer_barrier( aCmdBuff,
   aSceneUBO,
   VK_ACCESS_UNIFORM_READ_BIT,
   VK_ACCESS_TRANSFER_WRITE_BIT,
   VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,
   VK_PIPELINE_STAGE_TRANSFER_BIT
);

vkCmdUpdateBuffer( aCmdBuff, aSceneUBO, 0, sizeof(glsl::SceneUniform), &▽
  ▷ aSceneUniform );

lut::buffer_barrier( aCmdBuff,
   aSceneUBO,
   VK_ACCESS_TRANSFER_WRITE_BIT,
   VK_ACCESS_UNIFORM_READ_BIT,
   VK_PIPELINE_STAGE_TRANSFER_BIT,
   VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
);
```

Note the two buffer barriers. The first one ensures that the subsequent `vkCmdUpdateBuffer` only takes place after the vertex shader stage of any previous commands has completed. The uniform buffer is currently only accessed in the vertex shader; if it were used in both the vertex shader and fragment shader, both stages (`VK_PIPELINE_STAGE_VERTEX_INPUT_BIT|VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`) would have to be listed. The second barrier ensures that the results of the copy are visible to any subsequent uses of the buffer as a uniform buffer in the vertex shader stage.

Next, we need to bind the descriptor to specify the uniform inputs ahead of the draw call. Immediately after binding the graphics pipeline (`vkCmdBindPipeline`), use the `vkCmdBindDescriptorSets` function:

```
vkCmdBindDescriptorSets( aCmdBuff, VK_PIPELINE_BIND_POINT_GRAPHICS, aGraphicsLayout,▽
  ▷  0, 1, &aSceneDescriptors, 0, nullptr );
```

Run the program. You should see a result similar to the image shown in Figure 3.

4 Textures

(Tex.Coords — Texturing (GLSL) — Image creation — Image barriers — Tex. loading — Samplers — Descriptors — Drawing)

In this section, Exercise 1.4 introduces texturing. The overall structure is similar to that of the last section. We will first start by defining the correct input data, where we replace the per-vertex colors with texture coordinates. The texture coordinates are handled much like the vertex colors, in that the vertex shader simply copies the per-vertex texture coordinate input into a texture coordinate output variable. As with the colors, the texture coordinate is interpolated across the triangle during shader. The fragment shader receives the interpolated texture coordinate for the fragment, and uses it to sample a texture.

Aside from updating the input vertex buffers and shaders, texturing relies on a number of Vulkan objects. The texture data itself stems from an image file on disk, which must be loaded into a `VkImage` object. Exercise 1.4 enables mipmapping, meaning that each mipmap level needs to be defined. Further, a sampler object is required to describe how the texture is accesses (e.g., specifying filtering and addressing options). Both the image and sampler are passed to the shader via descriptors.

Texture coordinates The first step is to update the input data. Instead of using per-vertex colors, we wish to specify per-vertex texture coordinates. To do so, we replace the colors (3× `float`) with texture coordinates (2× `float`). The overall process is similar to the one in Section 3.

For clarity, we will not reuse the `ColorizedMesh`. Instead, define a `TexturedMesh` in `vertex_data.hpp`:

```
struct TexturedMesh
{
   labutils::Buffer positions;
   labutils::Buffer texcoords;

   std::uint32_t vertexCount;
};
```

We will then modify the `create_plane_mesh` function (you are free to create a new function, if you wish to keep the old one around). First, change the return type from `ColorizedMesh` to `TexturedMesh` both in the declaration (`vertex_data.hpp`) and definition (`vertex_data.cpp`):

```
TexturedMesh create_plane_mesh( labutils::VulkanContext const&, labutils::Allocator ▽
     ▷ const& );
```

Next, define the texture coordinates that replace the colors:

```
   static float const texcoord[] = {
      0.f, -6.f, // t0
      0.f, +6.f, // t1
      1.f, +6.f, // t2

      0.f, -6.f, // t0
      1.f, +6.f, // t2
      1.f, -6.f  // t3
   };
```

Update the rest of the method as necessary, replacing references to colors with texture coordinates. Do not forget to also update the return statement to return a `TexturedMesh` instead.

Texturing (GLSL) Similar to Section 3, we will use a new set of shaders for this exercise: `shaderTex.vert` and `shaderTex.frag`. Update `kVertShaderPath` and `kFragShaderPath` in `main.cpp` accordingly.

The vertex shader only requires minimal updates. Instead of receiving a `vec3 iColor` input and producing a `vec3 v2fColor` output, it will receive a `vec2 iTexCoord` input and produce a `vec2 v2fTexCoord` output:

```
#version 450

layout( location = 0 ) in vec3 iPosition;
layout( location = 1 ) in vec2 iTexCoord;

layout( set = 0, binding = 0 ) uniform UScene
{
   mat4 camera;
   mat4 projection;
   mat4 projCam;
} uScene;

layout( location = 0 ) out vec2 v2fTexCoord;

void main()
{
   v2fTexCoord = iTexCoord;

   gl_Position = uScene.projCam * vec4( iPosition, 1.f );
}
```

The main changes happen in the fragment shader. First, it will receive the `vec2 v2fTexCoord` instead of the `vec3 v2fColor`. Additionally, we somehow needs to identify the texture from which the shader should sample, and then perform the sampling.

Textures are uniform data, and passed to shaders with the mechanisms introduced in the previous section. Exercise 1.4 will use a separate descriptor set for the texture, as, conceptually, the texture changes with a higher frequency (once per material/object) than the per-frame transformation matrices. The following declares a `sampler2D` object named `uTexColor`. The sampler object identifies a texture:

```
layout( set = 1, binding = 0 ) uniform sampler2D uTexColor;
```

Samplers can be accessed using several GLSL built-in functions. The simplest is the `texture()` built-in. There are several different overloads. The one that we are using takes a `sampler2D` and a `vec2` argument, and returns a `vec4` with the texture data:

```
vec4 color = vec4( texture( uTexColor, v2fTexCoord );
```

The complete fragment shader follows:

```
#version 450

layout( location = 0 ) in vec2 v2fTexCoord;

layout( set = 1, binding = 0 ) uniform sampler2D uTexColor;

layout( location = 0 ) out vec4 oColor;

void main()
{
   oColor = vec4( texture( uTexColor, v2fTexCoord ).rgb, 1.f );
}
```

Image creation Texture data is held inside a `VkImage`. Exercise 1.2 introduced the "raw" Vulkan image creation. Similar to Section 2, we will now use VMA for this. Refer back to Section 2 for a detailed discussion. Analogous to the `labutils::Buffer` defined there, Exercise 1.4 introduces `labutils::Image` (see `labutils/vkimage.hpp` and `labutils/vkimage.cpp`). As with the new `Buffer`, the new `Image` replaces the raw Vulkan memory allocation (`VkDeviceMemory`) from Exercise 1.2 with a VMA `VmaAllocation` handle.

To create an image, VMA provides a `vmaCreateImage` function. The `vmaCreateImage` function is similar to the `vmaCreateBuffer` function that we have previously encountered. The main difference is that it takes a `VkImageCreateInfo` structure instead of the `VkBufferCreateInfo`. The `VkImageCreateInfo` was discussed in Exercise 1.2 – refer back to the previous exercise for details.

As Exercise 1.4 uses mipmapping, we must compute the number of mipmap levels for the `mipLevels` member of `VkImageCreateInfo`. The number of mipmap levels depends on the original image resolution. Recall the mipmap pyramid, where each level is half the size of the previous one along all dimensions (integer division). The smallest level is 1×1. We reach the smallest level by halving the larger image dimension `mipLevels` times. (Here, halving a dimension that is only one pixel in size remains at one pixel.)

The typical expression that one finds to compute `mipLevels` is as follows:

$$\text{mipLevels} = 1 + \left\lfloor \log_2 \left(\max \left(\text{width}, \text{height} \right) \right) \right\rfloor$$

Since we are dealing with image sizes that are integral, there is another way of viewing the problem. Each time we halve (divide by two) an integer, we shift its bits to the right (towards the less significant bits). The integer 1 corresponds exactly to a bit pattern where the least significant bit is set and all other bits are zero. We can take any integer and shift it $N - 1$ times to the right to produce this, where N is the index of the most significant set bit. (Any shifted-out bits are discarded.)

Finding the number N is very efficient, and a common enough operation that there are both hardware instructions to help with this (e.g., `LZCNT` on x86), as well as standard functions (e.g. `std::countl_zero`).

Unfortunately, `std::countl_zero` was introduced with C++20. Exercise 1.4 uses a portable fallback based on Bit Twiddling Hacks (an excellent resource for integer/bit manipulation-based techniques). Relying on integer-only math means that we do not have to think about rounding errors and similar floating point issues. Additionally, using the C++20 `std::countl_zero` (or a number of platform-specific intrinsics) simplifies the generated instructions significantly. You can compare the generated code (x86) for all three versions on Compiler Explorer (note that the assembly generated by the `log2`-version contains a function call to the standard `log2` function – meaning that not all executed code is shown!). But in this case, any gains in performance are unlikely to matter; after all computing numbers of mipmap levels is very unlikely to be a bottleneck here.

The `compute_mip_level_count` function declared in `labutils/vkimage.hpp` uses the latter technique to compute the number of mipmap levels given a certain image size. Find it in `labutils/vkimage.cpp`.

Implement `create_image_texture2d` in `vkimage.cpp` as follows:

```
auto const mipLevels = compute_mip_level_count( aWidth, aHeight );

VkImageCreateInfo imageInfo{};
imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageInfo.imageType       = VK_IMAGE_TYPE_2D;
imageInfo.format          = aFormat;
imageInfo.extent.width    = aWidth;
imageInfo.extent.height   = aHeight;
imageInfo.extent.depth    = 1;
imageInfo.mipLevels       = mipLevels;
imageInfo.arrayLayers     = 1;
imageInfo.samples         = VK_SAMPLE_COUNT_1_BIT;
imageInfo.tiling          = VK_IMAGE_TILING_OPTIMAL;
imageInfo.usage           = aUsage;
imageInfo.sharingMode     = VK_SHARING_MODE_EXCLUSIVE;
imageInfo.initialLayout   = VK_IMAGE_LAYOUT_UNDEFINED;

VmaAllocationCreateInfo allocInfo{};
allocInfo.flags   = 0;
allocInfo.usage   = VMA_MEMORY_USAGE_AUTO_PREFER_DEVICE;

VkImage image = VK_NULL_HANDLE;
VmaAllocation allocation = VK_NULL_HANDLE;

if( auto const res = vmaCreateImage( aAllocator.allocator, &imageInfo, &allocInfo, &
  ▷ image, &allocation, nullptr ); VK_SUCCESS != res )
{
   throw Error( "Unable to allocate image.\n"
      "vmaCreateImage() returned %s", to_string(res).c_str()
   );
}

return Image( aAllocator.allocator, image, allocation );
```

As with earlier uses of Vulkan images, an image view (`VkImageView`) is required in addition to the `VkImage`. Declare a function `create_image_view_texture2d` in `labutils/vkutil.hpp`:

```
ImageView create_image_view_texture2d( VulkanContext const&, VkImage, VkFormat );
```

Define the function in `labutils/vkutil.cpp` (refer back to Exercise 1.2 for details):

```
ImageView create_image_view_texture2d( VulkanContext const& aContext, VkImage aImage
  ▷ , VkFormat aFormat )
{
   VkImageViewCreateInfo viewInfo{};
   viewInfo.sType  = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
   viewInfo.image              = aImage;
   viewInfo.viewType           = VK_IMAGE_VIEW_TYPE_2D;
   viewInfo.format             = aFormat;
   viewInfo.components         = VkComponentMapping{}; // == identity
   viewInfo.subresourceRange   = VkImageSubresourceRange{
      VK_IMAGE_ASPECT_COLOR_BIT,
         0, VK_REMAINING_MIP_LEVELS,
         0, 1
      };

      VkImageView view = VK_NULL_HANDLE;
      if( auto const res = vkCreateImageView( aContext.device, &viewInfo, nullptr, &▽
    ▷ view ); VK_SUCCESS != res )
      {
         throw Error( "Unable to create image view\n"
            "vkCreateImageView() returned %s", to_string(res).c_str()
         );
      }

      return ImageView( aContext.device, view );
   }
```

Note the `VkImageSubresourceRange`. Previously, we set the `baseMipLevel` and `levelCount` members (second and third member) to 0 and 1, respectively. To use the mipmap levels, we now set the `baseMipLevel` to 0 and the `levelCount` to the special value `VK_REMAINING_MIP_LEVELS`. This indicates that all mipmap levels should be used when accessing the image through the image view.

Finally, create an image view in `main()` with the function:

```
   lut::ImageView floorView = lut::create_image_view_texture2d( window, floorTex.image,▽
     ▷  VK_FORMAT_R8G8B8A8_SRGB );
```

Image barriers Image barriers are similar to the buffer barriers introduced earlier. However, in addition to providing synchronization, image barriers are also used to transition the layout (Exercise 1.2) of images. Each operation that uses a Vulkan image requires the image to be in a certain layout.

So far, image layout transitions have occured as part of the render pass. Exercise 1.4 uses images outside of render passes (e.g., as textures), and we therefore cannot rely on this mechanism to transition layouts. Instead, we will use Vulkan's barriers to transition the image layout. The barriers additionally provide the necessary synchronization between consecutive operations on a image.

Image barriers are, like the buffer barriers, recorded into a command buffer with `vkCmdPipelineBarrier`. Image barriers are defined with the `VkImageMemoryBarrier` structure.

Like with the buffer barriers, Exercise 1.4 will use a helper function, `image_barrier`. Declare the function as follows in `labutils/vkutil.hpp`:

```
   void image_barrier(
      VkCommandBuffer,
      VkImage,
      VkAccessFlags aSrcAccessMask,
      VkAccessFlags aDstAccessMask,
      VkImageLayout aSrcLayout,
      VkImageLayout aDstLayout,
      VkPipelineStageFlags aSrcStageMask,
      VkPipelineStageFlags aDstStageMask,
      VkImageSubresourceRange = VkImageSubresourceRange{VK_IMAGE_ASPECT_COLOR_BIT▽
    ▷ ,0,1,0,1},
      std::uint32_t aSrcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED,
      std::uint32_t aDstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED
   );
```

Most of the arguments are shared with `buffer_barrier`. The major addition are the layouts. The *source* layout (`aSrcLayout`) specifies the layout that the image is expected to be in before the barrier. The *destination* layout (`aDstLayout`) specifies the layout to which the image should be transitioned by the barrier.

Add the definition of `image_barrier` to `labutils/vkutil.cpp`:

```
   void image_barrier( VkCommandBuffer aCmdBuff, VkImage aImage, VkAccessFlags ▽
    ▷ aSrcAccessMask, VkAccessFlags aDstAccessMask, VkImageLayout aSrcLayout, ▽
    ▷ VkImageLayout aDstLayout, VkPipelineStageFlags aSrcStageMask, ▽
    ▷ VkPipelineStageFlags aDstStageMask, VkImageSubresourceRange aRange, std::▽
    ▷ uint32_t aSrcQueueFamilyIndex, std::uint32_t aDstQueueFamilyIndex )
```

```
	{
		VkImageMemoryBarrier ibarrier{};
		ibarrier.sType                = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
		ibarrier.image                = aImage;
		ibarrier.srcAccessMask        = aSrcAccessMask;
		ibarrier.dstAccessMask        = aDstAccessMask;
		ibarrier.srcQueueFamilyIndex  = aSrcQueueFamilyIndex;
		ibarrier.dstQueueFamilyIndex  = aDstQueueFamilyIndex;
		ibarrier.oldLayout            = aSrcLayout;
		ibarrier.newLayout            = aDstLayout;
		ibarrier.subresourceRange     = aRange;

		vkCmdPipelineBarrier( aCmdBuff, aSrcStageMask, aDstStageMask, 0, 0, nullptr, 0, nullptr, 1, &ibarrier );
	}
```

Image loading We will implement image loading in the `load_image_texture2d` function in `vkimage.cpp`.

We will need to define all mipmap levels ourselves – unlike OpenGL, which has the `glGenerateMipmap` and `glGenerateTextureMipmap` functions, Vulkan does not provide such functionality out of the box. Here, we will utilize Vulkan's `vkCmdBlitImage` to compute each mipmap level on the fly. This may not create the most high quality mipmap images, but it is a very cheap and efficient solution.

The first step is to load the provided sample texture `asphalt.png` found in the `assets/exercise4` folder. For this, Exercise 1.4 uses the `stb_image.h` library. The library was introduced in Exercise 1.2 together with its counterpart `stb_image_write.h`. From `stb_image.h`, we use the following functions:

`stbi_load` Load image data from a file. Also returns basic information about the image, such as width, height and number of color channels. We can also specify the umber of color channels the loaded image should use. Returns an error if the file is unavailable or the image format is unsupported.

`stbi_failure_reason` On error, return a short description of the cause.

`stbi_set_flip_vertically_on_load` Determines if images should be flipped on load. Vulkan expects the first row/scanline of an image to be the bottom row; many image formats including PNG instead consider the first row/scanline to be the top one.

The overall process consists of the following steps:

1. Load the base image (e.g. `asphalt.png`) with `stbi_load`.
2. Create staging buffer with `create_buffer` (see buffer utilities).
3. Copy image data into a staging buffer (see buffer creation).
4. Create image with `create_image_texture2d` (see image creation).
5. Allocate command buffer and begin recording (see Exercise 1.2).
6. Transition entire image to the `TRANSFER_DST_OPTIMAL` layout (see image barriers).
7. Copy the staging buffer to base mip level (level 0) of the image with `vkCmdCopyBufferToImage`.
8. Transition base mipmap level's layout to `TRANSFER_SRC_OPTIMAL`
9. Compute number of mipmap levels, and for each level *except* the base level:
   (a) Use `vkCmdBlitImage` to blit the previous mipmap level's contents to the current one while also scaling it down by 50%. We use linear filtering (`VK_FILTER_LINEAR`) for this.
   (b) Transition mipmap level's layout to `TRANSFER_SRC_OPTIMAL` for next iteration
10. Transition the entire image to the `SHADER_READ_ONLY_OPTIMAL` layout.
11. End recording and submit commands for execution.
12. Wait for commands to finish before freeing temporary resources (staging buffer & command buffer)

One caveat is that the staging buffer needs to remind valid until after the commands have finished executing. We therefore need to be careful with the buffer object's lifetime.

Find the definition of `load_image_texture2d` in `labutils/vkimage.cpp`. As outlined above, the first step is to load the base image:

```
// Flip images vertically by default. Vulkan expects the first scanline to be the bottom-most scanline. PNG et al.
// instead define the first scanline to be the top-most one.
stbi_set_flip_vertically_on_load( 1 );

// Load base image
int baseWidthi, baseHeighti, baseChannelsi;
stbi_uc* data = stbi_load( aPath, &baseWidthi, &baseHeighti, &baseChannelsi, 4 /* want 4 channels = RGBA */ );

if( !data )
{
   throw Error( "%s: unable to load texture base image (%s)", aPath, 0, stbi_failure_reason() );
}

auto const baseWidth = std::uint32_t(baseWidthi);
auto const baseHeight = std::uint32_t(baseHeighti);
```

Aside from the image data, we also learn the texture's base extents.

We immediately transfer the data into a staging buffer:

```
// Create staging buffer and copy image data to it
auto const sizeInBytes = baseWidth * baseHeight * 4;

auto staging = create_buffer( aAllocator, sizeInBytes, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT );

void* sptr = nullptr;
if( auto const res = vmaMapMemory( aAllocator.allocator, staging.allocation, &sptr ); VK_SUCCESS != res )
{
   throw Error( "Mapping memory for writing\n"
      "vmaMapMemory() returned %s", to_string(res).c_str()
   );
}

std::memcpy( sptr, data, sizeInBytes );
vmaUnmapMemory( aAllocator.allocator, staging.allocation );

// Free image data
stbi_image_free( data );
```

After copying to the staging buffer, we no longer need the original copy of the image that `stbi_load` returned. The code therefore frees the memory with `stbi_image_free`.

There is potentially some room for a minor optimization here. If we were able to decompress the image immediately into the memory of the Vulkan staging buffer, it would save us an allocation and a copy. However, the `stb_image.h` library does not provide an interface for this. We would also need to be careful about how the memory is accessed – e.g., would it need to be `HOST_CACHED`?

Next, we create the image with the previously defined `create_image_texture2d` function:

```
// Create image
Image ret = create_image_texture2d( aAllocator, baseWidth, baseHeight, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_USAGE_SAMPLED_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_TRANSFER_SRC_BIT );
```

We fix the format to `VK_FORMAT_R8G8B8A8_SRGB` and declare the intentions to use the image as a sampled texture (`VK_IMAGE_USAGE_SAMPLED_BIT`) and to copy data to it (`VK_IMAGE_USAGE_TRANSFER_DST_BIT`) and copy data from it with the blit (`VK_IMAGE_USAGE_TRANSFER_SRC_BIT`).

The following processing takes place via Vulkan commands, which requires a command buffer to record them into. We create one:

```
// Create command buffer for data upload and begin recording
VkCommandBuffer cbuff = alloc_command_buffer( aContext, aCmdPool );

VkCommandBufferBeginInfo beginInfo{};
beginInfo.sType  = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
beginInfo.flags             = 0;
beginInfo.pInheritanceInfo  = nullptr;

if( auto const res = vkBeginCommandBuffer( cbuff, &beginInfo ); VK_SUCCESS != res )
{
   throw Error( "Beginning command buffer recording\n"
      "vkBeginCommandBuffer() returned %s", to_string(res).c_str()
   );
}
```

In order to be able to copy data to a mipmap level, it needs to be in the `TRANSFER_DST_OPTIMAL` layout. We will copy data to each mipmap level (either via `vkCmdCopyBufferToImage` or via `vkCmdBlitImage`), so transitioning the entire image (with all levels) to the layout up-front is a good option:

```
// Transition whole image layout
// When copying data to the image, the image's layout must be TRANSFER_DST_OPTIMAL. The current
// image layout is UNDEFINED (which is the initial layout the image was created in).
auto const mipLevels = compute_mip_level_count( baseWidth, baseHeight );

image_barrier( cbuff, ret.image,
   0,
   VK_ACCESS_TRANSFER_WRITE_BIT,
   VK_IMAGE_LAYOUT_UNDEFINED,
   VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
   VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
   VK_PIPELINE_STAGE_TRANSFER_BIT,
   VkImageSubresourceRange{
      VK_IMAGE_ASPECT_COLOR_BIT,
      0, mipLevels,
      0, 1
   }
);
```

The copy can now be issued:

```
// Upload data from staging buffer to image
VkBufferImageCopy copy;
copy.bufferOffset       = 0;
copy.bufferRowLength    = 0;
copy.bufferImageHeight  = 0;
copy.imageSubresource   = VkImageSubresourceLayers{
   VK_IMAGE_ASPECT_COLOR_BIT,
   0,
   0, 1
};
copy.imageOffset        = VkOffset3D{ 0, 0, 0 };
copy.imageExtent        = VkExtent3D{ baseWidth, baseHeight, 1 };

vkCmdCopyBufferToImage( cbuff, staging.buffer, ret.image, ▽
  ▷ VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &copy );
```

The next operation will read from the base level and blit into the next smaller mipmap level. For this operation, the base level must be in the `TRANSFER_SRC_OPTIMAL` layout. The image barrier further guarantees that the blit sees the results of the copy before starting:

```
// Transition base level to TRANSFER_SRC_OPTIMAL
image_barrier( cbuff, ret.image,
   VK_ACCESS_TRANSFER_WRITE_BIT,
   VK_ACCESS_TRANSFER_READ_BIT,
   VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
   VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
   VK_PIPELINE_STAGE_TRANSFER_BIT,
    VK_PIPELINE_STAGE_TRANSFER_BIT,
    VkImageSubresourceRange{
       VK_IMAGE_ASPECT_COLOR_BIT,
       0, 1,
       0, 1
    }
);
```

Subsequent operation will fill in the remaining mipmap levels. For each level, we blit data out of the previous level. Unlike a straight copy, the blit can scale the image. Here, we scale down the image by approximately 50% each time. Finally, the mipmap level is transitioned to the `TRANSFER_SRC_OPTIMAL` layout, so that the next level's blit can use the current level as a source (and to ensure proper synchronization):

```
// Process all mipmap levels
uint32_t width = baseWidth, height = baseHeight;

for( std::uint32_t level = 1; level < mipLevels; ++level )
{
   // Blit previous mipmap level (=level-1) to the current level. Note that the loop starts at level = 1.
   // Level = 0 is the base level that we initialied before the loop.
   VkImageBlit blit{};
   blit.srcSubresource = VkImageSubresourceLayers{
      VK_IMAGE_ASPECT_COLOR_BIT,
      level-1,
      0, 1
   };
   blit.srcOffsets[0] = { 0, 0, 0 };
   blit.srcOffsets[1] = { std::int32_t(width), std::int32_t(height), 1 };

   // Next mip level
   width >>= 1; if( width == 0 ) width = 1;
   height >>= 1; if( height == 0 ) height = 1;

   blit.dstSubresource = VkImageSubresourceLayers{
      VK_IMAGE_ASPECT_COLOR_BIT,
      level,
      0, 1
   };
   blit.dstOffsets[0] = { 0, 0, 0 };
   blit.dstOffsets[1] = { std::int32_t(width), std::int32_t(height), 1 };

   vkCmdBlitImage( cbuff,
      ret.image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
      ret.image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
      1, &blit,
      VK_FILTER_LINEAR
   );

   // Transition mip level to TRANSFER_SRC_OPTIMAL for the next iteration. (Technically this is
   // unnecessary for the last mip level, but transitioning it as well simplifes the final barrier following the
   // loop).
   image_barrier( cbuff, ret.image,
      VK_ACCESS_TRANSFER_WRITE_BIT,
      VK_ACCESS_TRANSFER_READ_BIT,
      VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
      VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
      VK_PIPELINE_STAGE_TRANSFER_BIT,
      VK_PIPELINE_STAGE_TRANSFER_BIT,
      VkImageSubresourceRange{
         VK_IMAGE_ASPECT_COLOR_BIT,
         level, 1,
         0, 1
      }
   );
}
```

It might be tempting to always copy out of the base level. However, with the simple linear filter supported by `vkCmdBlitImage`, this is not possible and will lead to significant aliasing. The linear filter only considers a 2×2 kernel. Even just going down two levels means that the resolution is reduced to one quarter, and subsequently each new texel would be influenced by a 4×4 region (box filter). The simple linear filter misses this, and thus discards data instead of filtering it. A filter with a larger footprint would be required, but `vkCmdBlitImage` does not support this by default.

The final command transitions the image to `SHADER_READ_ONLY_OPTIMAL` such that we can sample from it:

```
// Whole image is currently in the TRANSFER_SRC_OPTIMAL layout. To use the image as a texture from
// which we sample, it must be in the SHADER_READ_ONLY_OPTIMAL layout.
image_barrier( cbuff, ret.image,
   VK_ACCESS_TRANSFER_READ_BIT,
   VK_ACCESS_SHADER_READ_BIT,
   VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
   VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
   VK_PIPELINE_STAGE_TRANSFER_BIT,
   VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
   VkImageSubresourceRange{
      VK_IMAGE_ASPECT_COLOR_BIT,
      0, mipLevels,
      0, 1
   }
);
```

This completes the mipmap generation. Submit the commands for execution, ensure that they have completed, and then release temporary resources:

```
// End command recording
if( auto const res = vkEndCommandBuffer( cbuff ); VK_SUCCESS != res )
{
   throw Error( "Ending command buffer recording\n"
      "vkEndCommandBuffer() returned %s", to_string(res).c_str()
   );
}

// Submit command buffer and wait for commands to complete. Commands must have completed before we can
// destroy the temporary resources, such as the staging buffers.
Fence uploadComplete = create_fence( aContext );

VkSubmitInfo submitInfo{};
submitInfo.sType  = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount    = 1;
submitInfo.pCommandBuffers       = &cbuff;

if( auto const res = vkQueueSubmit( aContext.graphicsQueue, 1, &submitInfo, ▽
  ▷ uploadComplete.handle ); VK_SUCCESS != res )
{
   throw Error( "Submitting commands\n"
      "vkQueueSubmit() returned %s", to_string(res).c_str()
   );
}

if( auto const res = vkWaitForFences( aContext.device, 1, &uploadComplete.handle, ▽
  ▷ VK_TRUE, std::numeric_limits<std::uint64_t>::max() ); VK_SUCCESS != res )
{
   throw Error( "Waiting for upload to complete\n"
      "vkWaitForFences() returned %s", to_string(res).c_str()
   );
}

// Return resulting image
// Most temporary resources are destroyed automatically through their destructors. However, the command
// buffer we must free manually.
vkFreeCommandBuffers( aContext.device, aCmdPool, 1, &cbuff );

return ret;
```

To keep the `load_image_texture2d` function somewhat simple and self-contained, the method waits for the Vulkan commands to execute. This is likely somewhat suboptimal. A better strategy would be to submit the commands for processing and, for example, start with loading the next texture immediately. However, doing so complicates the implementation somewhat, as some resources (staging and command buffers) now have a lifetime longer than the function itself. The exercise therefore opts for the simpler (but less performant) option.

In `main()`, use `load_image_texture2d` to load the floor texture:

```
lut::Image floorTex;

{
   lut::CommandPool loadCmdPool = lut::create_command_pool( window, ▽
  ▷ VK_COMMAND_POOL_CREATE_TRANSIENT_BIT );

   floorTex = lut::load_image_texture2d( cfg::kFloorTexture, window, loadCmdPool.▽
  ▷ handle, allocator );
}
```

(The extra scope is introduced to limit the lifetime of the command pool to just texture loading. If we were to load multiple textures, we would want to minimally reuse the same command pool for all textures.)

Sampler While the texture image data is held in a `VkImage`, the `VkSampler` object defines how the texture is sampled. Creating a `VkSampler` is comparatively straight forward, and just uses

- the `vkCreateSampler` function and
- the `VkSamplerCreateInfo` structure.

The relevant fields of the `VkSamplerCreateInfo` structure are:

`magFilter`, `minFilter` Magnification and minification filters. Possible values are defined by the `VkFilter` enumeration. Common values are either `VK_NEAREST` for nearest-neighbour filtering or `VK_LINEAR` for linear filter. We will default to `VK_LINEAR`.

`mipmapMode` Mipmapping mode. One of the `VkSamperMipmapMode` constants, which defines two options: either `VK_SAMPLER_MIPMAP_MODE_NEAREST` or `VK_SAMPLER_MIPMAP_MODE_LINEAR`. We select the latter (`VK_SAMPLER_MIPMAP_MODE_LINEAR`) by default, resulting in trilinear filtering when combined with the `VK_LINEAR` minification filter above.

`addressModeU`, `addressModeV` Addressing mode. See the `VkSamplerAddressMode` for possible choices. We will use `VK_SAMPLER_ADDRESS_MODE_REPEAT` in Exercise 1.4, as this allows us to repeat the floor texture several times across the plane geometry.

`minLod`, `maxLod` Determines minimum and maximum mipmap LOD value. We do not want to resitrict the mipmap LOD selection, and consequently set these to 0 and `VK_LOD_CLAMP_NONE`, respectively. This causes the sampler to use all the available mipmap levels.

`mipLodBias` Affects the mipmap LOD computation. Setting this to a positive value shifts mipmap selection towards lower resolution mipmap levels, and a negative value shifts mipmap selection to higher resolution images. We leave it at 0 for now.

The remaining options are left initialized to zero for the time being.

Declare a `create_default_sampler` function in `labutils/vkutil.hpp`:

```
Sampler create_default_sampler( VulkanContext const& );
```

Define the function in `labutils/vkutil.cpp`:

```
Sampler create_default_sampler( VulkanContext const& aContext )
{
   VkSamplerCreateInfo samplerInfo{};
   samplerInfo.sType   = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
   samplerInfo.magFilter      = VK_FILTER_LINEAR;
   samplerInfo.minFilter      = VK_FILTER_LINEAR;
   samplerInfo.mipmapMode     = VK_SAMPLER_MIPMAP_MODE_LINEAR;
   samplerInfo.addressModeU   = VK_SAMPLER_ADDRESS_MODE_REPEAT;
   samplerInfo.addressModeV   = VK_SAMPLER_ADDRESS_MODE_REPEAT;
   samplerInfo.minLod         = 0.f;
```

```
    samplerInfo.maxLod          = VK_LOD_CLAMP_NONE;
    samplerInfo.mipLodBias      = 0.f;

    VkSampler sampler = VK_NULL_HANDLE;
    if( auto const res = vkCreateSampler( aContext.device, &samplerInfo, nullptr, &▽
   ▷ sampler ); VK_SUCCESS != res )
    {
       throw Error( "Unable to create sampler\n"
          "vkCreateSampler() returned %s", to_string(res).c_str()
       );
    }

    return Sampler( aContext.device, sampler );
}
```

Finally, create the sampler object in `main()` after loading the floor texture:

```
lut::Sampler defaultSampler = lut::create_default_sampler( window );
```

Descriptors Setting up the descriptors mirrors the process from Section 3. As Exercise 1.4 elects to use a separate descriptor set, we must first define layout for it. The layout, like the one in Section 3, is quite simple: it has a single descriptor. The descriptor's type is `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`. As the name indicates, this combines a sampler (`VkSampler`) with an image (`VkImage`). Uniform inputs with the GLSL type `sampler2D` correspond to `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`-descriptors and vice-versa.

Locate the definition of `create_object_descriptor_layout` in `main.cpp`. The implementation will be very similar to `create_scene_descriptor_layout`. The list of `VkDecriptorSetLayoutBinding`s changes slightly:

```
VkDescriptorSetLayoutBinding bindings[1]{};
bindings[0].binding           = 0; // this must match the shaders
bindings[0].descriptorType    = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
bindings[0].descriptorCount   = 1;
bindings[0].stageFlags        = VK_SHADER_STAGE_FRAGMENT_BIT;

VkDescriptorSetLayoutCreateInfo layoutInfo{};
layoutInfo.sType          = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount   = sizeof(bindings)/sizeof(bindings[0]);
layoutInfo.pBindings      = bindings;

VkDescriptorSetLayout layout = VK_NULL_HANDLE;
if( auto const res = vkCreateDescriptorSetLayout( aWindow.device, &layoutInfo, ▽
  ▷ nullptr, &layout ); VK_SUCCESS != res )
{
   throw lut::Error( "Unable to create descriptor set layout\n"
      "vkCreateDescriptorSetLayout() returned %s", lut::to_string(res).c_str()
   );
}

return lut::DescriptorSetLayout( aWindow.device, layout );
```

In addition to using the aforementioned `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, the `stageFlags` member is changed to `VK_SHADER_STAGE_FRAGMENT_BIT`. This reflects the fact that the texture is used in the fragment shader (and not in the vertex shader, like the uniform buffer in Section 3).

Create the "per-object" descriptor set layout in `main()`, next to the "scene" descriptor set layout:

```
lut::DescriptorSetLayout objectLayout = create_object_descriptor_layout( window );
```

Like the `sceneLayout`, we need to pass the new `objectLayout` to the `create_pipeline_layout` method. Update the declaration of `create_pipeline_layout` to accept an additional `VkDescriptorSetLayout` argument, and pass the newly created `objectLayout.handle` to `create_pipeline_layout` when called in `main()`. Finally, update the definition of `create_triangle_pipeline`:

```
lut::PipelineLayout create_pipeline_layout( lut::VulkanContext const& aContext, ▽
  ▷ VkDescriptorSetLayout aSceneLayout, VkDescriptorSetLayout aObjectLayout )
{
   VkDescriptorSetLayout layouts[] = {
        // Order must match the set = N in the shaders
        aSceneLayout,
        aObjectLayout
    };

    // ...
```

Recall that the order in which the `VkDescriptorSetLayout`s are passed to `vkCreatePipelineLayout` determine which GLSL set (`set = N`) they correspond to. A brief check with the shaders should verify that this is indeed the correct order: the first element, the scene descriptor set layout with its single uniform buffer, corresponds to the uniform inputs with `set = 0`; the second element, the object descriptor set layout the texture, corresponds to the uniform inputs with `set = 1`.

As with the scene descriptor set, the final step is to allocate a `VkDescriptorSet` and initialize it with the correct data (see descriptor sets). We can reuse the descriptor pool that was previously recreated:

```
VkDescriptorSet floorDescriptors = lut::alloc_desc_set( window, dpool.handle, ▽
  ▷ objectLayout.handle );

{
   VkWriteDescriptorSet desc[1]{};

   VkDescriptorImageInfo textureInfo{};
   textureInfo.imageLayout  = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
   textureInfo.imageView    = floorView.handle;
   textureInfo.sampler      = defaultSampler.handle;

   desc[0].sType  = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
   desc[0].dstSet           = floorDescriptors;
   desc[0].dstBinding       = 0;
   desc[0].descriptorType   = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
   desc[0].descriptorCount  = 1;
   desc[0].pImageInfo       = &textureInfo;


   constexpr auto numSets = sizeof(desc)/sizeof(desc[0]);
   vkUpdateDescriptorSets( window.device, numSets, desc, 0, nullptr );
}
```

As we are now defining a descriptor of type `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`, we pass the information about the image and sampler via the `pImageInfo` member of `VkWriteDescriptorSet` (we previously used the `pBufferInfo` for uniform buffer descriptor). In the `VkDescriptorImageInfo` structure we specify the image view to the texture and the previously created sampler object. Vulkan housekeeping further requires us to specify the layout of the image at the time the image is accessed through the descriptor. In our case, the layout is simply `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, to which we transitioned the image after initializing it with data.

Drawing To make the texture available to the fragment shader when drawing the plane mesh, we need to bind the descriptor that we just created. The process is similar to that in Section 3.

First, add another argument to `record_commands`:

```
void record_commands(
   VkCommandBuffer,
   VkRenderPass,
   VkFramebuffer,
   VkPipeline,
   VkExtent2D const&,
   VkBuffer aPositionBuffer,
   VkBuffer aTexCoordBuffer, // Name changed!
   std::uint32_t aVertexCount,
   VkBuffer aSceneUBO,
   glsl::SceneUniform const&,
   VkPipelineLayout,
   VkDescriptorSet aSceneDescriptors,
   VkDescriptorSet aObjectDescriptors
);
```

(rename the `aColorBuffer` argument to `aTexCoordBuffer` to more accurately reflect what it is used for, if you have not done so already).

Call `record_commands` in `main()` with the appropriate arguments:

```
record_commands(
    cbuffers[imageIndex],
    renderPass.handle,
    framebuffers[imageIndex].handle,
    pipe.handle,
    window.swapchainExtent,
    planeMesh.positions.buffer,
    planeMesh.texcoords.buffer,
    planeMesh.vertexCount,
    sceneUBO.buffer,
    sceneUniforms,
    pipeLayout.handle,
    sceneDescriptors,
    floorDescriptors
);
```

Update the definition of `record_commands`. Rename the `aColorBuffer` argument to `TexCoordBuffer` for consistency. The add a second call to `vkCmdBindDescriptorSets`:

```
vkCmdBindDescriptorSets( aCmdBuff, VK_PIPELINE_BIND_POINT_GRAPHICS, aGraphicsLayout, ↘
  ▷  1, 1, &aObjectDecriptors, 0, nullptr );
```

As `vkCmdBindDescriptorSets` allows us to bind multiple descriptor sets in a single call, it is possible to combine the two calls (one binding the `aSceneDescriptors` and the second one binding `aObjectDescriptors`) into a single call. You may do so if you wish. Keeping the two separate illustrates the use case where the `aObjectDescriptors` may change between different draw calls, whereas the `aSceneDescriptors` would remain the same.

Wrapping up

At this point, you should be able to build and run the program. The resulting image should look like the one shown in the teaser image. Note in particular the overblur in the middle-stripe towards the far end. This is a common artifact when using mipmapping.

As before, make sure the application is not producing any untoward validation errors. Use the `vkconfig` / Vulkan Configurator tool to check for e.g. synchronization errors. Refer to Exercise 3 for details.

The next exercise will add a few standard features that we are still missing. In particular, it will add a depth buffer for hidden surface removal. It will also demonstrate alpha blending and the use of an additional graphics pipeline to render different "materials".

Optional experiments

There are a few optional experiments that you can conduct:

Anisotropic filtering The current setup uses mipmaps with tri-linear filtering, which results in noticeable overblurring. Anisotropic filtering will reduce this. To enable anisotropic filtering, two steps are required. First, anisotropic filtering must be enabled during *device creation*, via the `VkPhysicalDeviceFeatures` structure. Although it is supported by most Vulkan implementations, one should nevertheless check this by querying what features the (physical) device supports. Secondly, anisotropic filtering needs to be enabled in the `VkSampler` object. The amount of anisotropic filtering can be tweaked when creating the sampler. More filtering is more expensive. Each device defines a maximum amount it can support (which can also be queried from the physical device).

Indexed meshes Exercise 1.4 renders a "triangle soup" with `vkCmdDraw`. A more efficient approach is to use indexed meshes with `vkCmdDrawIndexed`. Indexed rendering requires another buffer, a index buffer (`VK_BUFFER_USAGE_INDEX_BUFFER_BIT`). The index buffer is bound with `vkCmdBindIndexBuffer` ahead of the call to `vkCmdDrawIndexed`. (For the small meshes presented herein the difference might not be substantial, but for larger meshes it will be appreciable.)

Acknowledgements
The document uses icons from https://icons8.com: , , . . The "free" license requires attribution in documents that use the icons. Note that each icon is a link to the original source thereof.