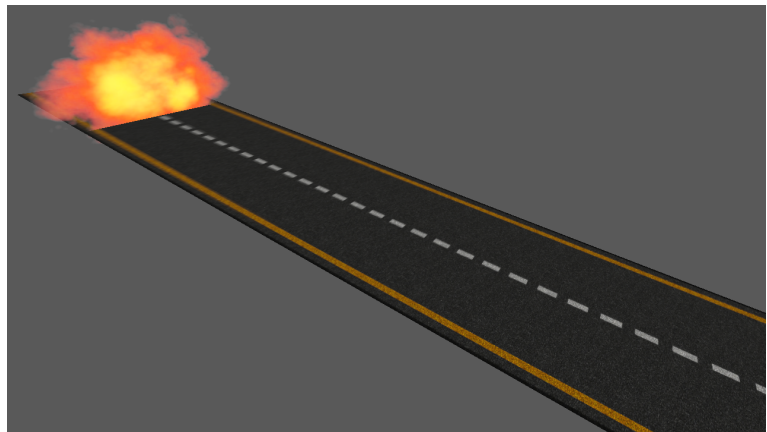


COMP5822M – Exercise 1.5

Depth testing & Pipelines

Contents

1	Second object	1
2	Depth buffer & testing	4
3	2nd pipeline & Blending	8
4	User Camera	10



Exercise 5 wraps up the Vulkan-introduction and ties up a few loose ends. The most important step in Exercise 5 is the addition of a depth buffer for hidden surface removal. The depth buffer is a key element in almost all 3D projective graphics. In addition, Exercise 5 demonstrates the use of multiple graphics pipelines. The two sample objects in Exercise 5 differ in that one is fully opaque and the other one is made transparent/translucent with alpha blending. Finally, it also demonstrates how to implement a 3D camera, which is a useful tool for debugging.

Exercise 5 builds directly on Exercise 4. You can just copy over your Exercise 4 to Exercise 5 (or work directly on Exercise 4). No new third party software is required; and no new `labutils` elements are added.

1 Second object

We will start by introducing a second object into the mix. With all the code that is already in place, this is (relatively speaking) painless.

Like with the earlier objects, we will simply define the geometry in a C++ array. However, you can likely see at this point how this hand-written array could easily be replaced with data loaded from a file.

The geometry is again a quad, but this time facing the initial default camera. First, declare an additional function `create_sprite_mesh` in `vertex_data.hpp`:

```
TexturedMesh create_sprite_mesh( labutils::VulkanContext const&, labutils::Allocator ∇ 1  
    ▷ const& );
```

Implement the function in `vertex_data.cpp`. Aside from the pre-defined mesh data, the function will be more or less identical to `create_plane_mesh`:

```
TexturedMesh create_sprite_mesh( labutils::VulkanContext const& aContext, labutils::∇ 1  
    ▷ Allocator const& aAllocator )  
{  
    // Vertex data  
    static float const positions[] = {  
        1  
        2  
        3  
        4
```

```

-1.5f, +1.5f, -4.f, // v0
-1.5f, -0.5f, -4.f, // v1
+1.5f, -0.5f, -4.f, // v2

-1.5f, +1.5f, -4.f, // v0
+1.5f, -0.5f, -4.f, // v2
+1.5f, +1.5f, -4.f // v3
};

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

    0.f, 1.f, // t0
    1.f, 0.f, // t2
    1.f, 1.f // t3
};

// ...

```

You may choose to simply create (another) copy of the function's body, or, if you want to, factor out the common parts into a separate reusable helper function.

Call the function in `main()`, next to the existing call to `create_plane_mesh`:

```
TexturedMesh spriteMesh = create_sprite_mesh( window, allocator );
```

The new object will use a different texture, using the `bricks.png` base image. In `main.cpp`, next to the definition of `kFloorTexture`, add:

```
constexpr char const* kSpriteTexture = ASSERTDIR_ "bricks.png";
```

In `main()`, use `load_image_texture2d` to load the images into a Vulkan image and create the corresponding image view:

```

// ... existing code defining the floorTex variable ...
lut::Image spriteTex;

{
    // ... existing code creating the loadCmdPool and loading the floor texture ...

    spriteTex = lut::load_image_texture2d( cfg::kSpriteTexture, window, loadCmdPool.▽
▷ handle, allocator );
}

// ... existing code creating the floorView image view
lut::ImageView spriteView = lut::create_image_view_texture2d( window, spriteTex.▽
▷ image, VK_FORMAT_R8G8B8A8_SRGB );

```

We can simply reuse the existing `VkSampler` object that was created for the floor texture.

Next, we will create a `VkDescriptorSet` that refers to the texture for this additional object. This descriptor set is of the same type as the descriptor set that we created in Exercise 4 (Section 4). This type is identified by the descriptor set layout (`VkDescriptorSetLayout`) that we created there, so we will need to reuse that exact layout when creating the new `VkDescriptorSet` instance:

```

VkDescriptorSet spriteDescriptors = 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 = spriteView.handle;
    textureInfo.sampler = defaultSampler.handle;
}

```

```

    desc[0].sType   = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
    desc[0].dstSet   = spriteDescriptors;
    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 );
}

```

Aside from using the `spriteView` image view, the code is identical to the one we used to initialize the descriptor set with the floor texture.

The final step is to record the commands for drawing the new object. To facilitate this, `record_commands` will need to take another set of arguments, specifically two `VkBuffer` handles for the new object's positions and texture coordinates, the number of vertices, and a `VkDescriptorSet` handle for the new descriptor set:

```

void record_commands(
    VkCommandBuffer,
    VkRenderPass,
    VkFramebuffer,
    VkPipeline,
    VkExtent2D const&
    VkBuffer aPositionBuffer,
    VkBuffer aTexCoordBuffer,
    std::uint32_t aVertexCount,
    VkBuffer aSceneUBO,
    glsl::SceneUniform const&,
    VkPipelineLayout,
    VkDescriptorSet aSceneDescriptors,
    VkDescriptorSet aObjectDescriptors,
    VkBuffer aSpritePosBuffer,
    VkBuffer aSpriteTexBuffer,
    std::uint32_t aSpriteVertexCount,
    VkDescriptorSet aSpriteObjDescriptors
);

```

Yes, this is getting somewhat out of hand. And indeed, we should be thinking about a neater way of passing around the information about our objects. However, this will be left up to you in e.g. Coursework 1. In this case, the purpose of the exercises is to let you experiment with Vulkan, without having to worry too much about neat software design. You can then use this experience to make an informed decision on the software design/architecture in future projects.



Make sure to update the call `record_commands` in `main()` and pass it the correct values.

Finally, add the new arguments to the definition of `record_commands`. In the body of `record_commands`, add the necessary draw calls to draw the new object (just after the existing `vkCmdDraw`, but before ending the render pass with `vkCmdEndRenderPass`):

```

vkCmdBindDescriptorSets( aCmdBuff, VK_PIPELINE_BIND_POINT_GRAPHICS, aGraphicsLayout, 0,
    1, 1, &aSpriteObjDescriptors, 0, nullptr );

VkBuffer spriteBuffers[2] = { aSpritePosBuffer, aSpriteTexBuffer };
VkDeviceSize spriteOffsets[2]{};

vkCmdBindVertexBuffers( aCmdBuff, 0, 2, spriteBuffers, spriteOffsets );

vkCmdDraw( aCmdBuff, aSpriteVertexCount, 1, 0, 0 );

```

Run the program. You should see results similar to Figure 1.

You can briefly experiment with reversing the order of the draw calls. Does that change the results? Should it?

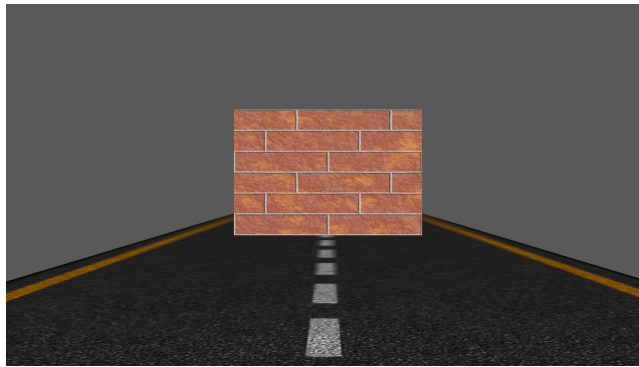


Figure 1: Second object, a brick wall, draw on top of the floor plane (Section 1).

2 Depth buffer & testing

(Depth buffer image creation — Framebuffers — Render Pass — Graphics Pipeline (depth testing) — Drawing (clear values))

Looking the results of the previous section (Figure 1), something is clearly not quite right yet. And, indeed, we are missing depth buffering/testing. We will now implement this.

While we have multiple swap chain images for different frames, a single depth buffer image is sufficient. Multiple swap chain images are required, as we might be rendering to one while a different one is being presented. This is not the case with the depth buffer. Here, after rendering, the depth buffer is no longer required. In this simple setup, there is also little point in attempting to render multiple frames concurrently. We'd rather just finish the current frame as fast as possible before starting a new one.



Depth buffer image Enabling depth testing requires us to provide a depth buffer. The depth buffer is just another `VkImage`, created with one of the depth-related `VkFormats`. For Exercise 1.4, we will simply use the `VK_FORMAT_D32_SFLOAT` format. This format seems to be almost universally supported on desktop-level hardware. You can check GPUInfo.org's list for [optimal tiling format support](#): `D32_SFLOAT` is supported universally desktop platforms (Windows, Linux and MacOS). Only on Android are there still some devices that do not support it.

If you are optimizing your rendering, it is always a good idea to check various vendors' best practice recommendations. For example, [NVIDIA](#) recommends using 24-bit depth formats for optimal performance (rather than the 32-bit float depth buffer the above opts for). [AMD](#) instead states that 24-bit and 32-bit depth formats have the same costs, but that 32-bit formats get better precision for the same memory cost (AMD further explicitly recommends using reverse-Z, which makes most sense with a floating point depth format). [Intel's](#) advice for their *Arc/Xe*-based graphics is just to use depth-only formats, including `D32`, rather than combined depth-stencil.



In the `cfg` namespace in `main.cpp`, add:

```
constexpr VkFormat kDepthFormat = VK_FORMAT_D32_SFLOAT;
```

1

Creation of the depth buffer image and its image view should be implemented in the `create_depth_buffer` function. Its declaration is

```
std::tuple<lut::Image, lut::ImageView> create_depth_buffer( lut::VulkanWindow const&, ▽
    ▹ lut::Allocator const& );
```

Find it already in `main.cpp` if you use the Exercise 5 code base, or add the declaration yourself if you are continuing from your Exercise 4' solutions. Review the image creation process with VMA (Exercise 4). Aside from having a depth-`VkFormat`, the usage flag `VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT` must be set (otherwise it will not be usable as a depth buffer). Further, the depth buffer's size must match the size of the swap chain:

```
VkImageCreateInfo imageInfo{};
imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageInfo.imageType = VK_IMAGE_TYPE_2D;
imageInfo.format = cfg::kDepthFormat;
imageInfo.extent.width = aWindow.swapchainExtent.width;
```

1

2

3

4

5

```

imageInfo.extent.height = aWindow.swapchainExtent.height;
imageInfo.extent.depth  = 1;
imageInfo.mipLevels      = 1;
imageInfo.arrayLayers    = 1;
imageInfo.samples        = VK_SAMPLE_COUNT_1_BIT;
imageInfo.tiling         = VK_IMAGE_TILING_OPTIMAL;
imageInfo.usage          = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
imageInfo.sharingMode    = VK_SHARING_MODE_EXCLUSIVE;
imageInfo.initialLayout  = VK_IMAGE_LAYOUT_UNDEFINED;

VmaAllocationCreateInfo allocInfo{};
allocInfo.usage = VMA_MEMORY_USAGE_GPU_ONLY;

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 lut::Error( "Unable to allocate depth buffer image.\n"
        "vmaCreateImage() returned %s", lut::to_string(res).c_str()
    );
}

lut::Image depthImage( aAllocator.allocator, image, allocation );

// Create the image view
VkImageViewCreateInfo viewInfo{};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
viewInfo.image = depthImage.image;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = cfg::kDepthFormat;
viewInfo.components = VkComponentMapping{};
viewInfo.subresourceRange = VkImageSubresourceRange{
    VK_IMAGE_ASPECT_DEPTH_BIT,
    0, 1,
    0, 1
};

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

return { std::move(depthImage), lut::ImageView( aWindow.device, view ) };

```

The image view's format must match the depth image's format. Additionally, the `aspectMask` member in the `VkImageSubresourceRange` is set to `VK_IMAGE_ASPECT_DEPTH_BIT` instead of `COLOR_BIT`.

Insert a call to `create_depth_buffer` in `main()`, just before initially creating the framebuffer objects with `create_swapchain_framebuffers`:

```
auto [depthBuffer, depthBufferView] = create_depth_buffer( window, allocator );
```

Since the depth buffer must match the size of the framebuffer, it must be recreated whenever the window's size changes. Do so just before recreating the framebuffers in response to changes to the swap chain:

```
if( changes.changedSize )
    std::tie(depthBuffer, depthBufferView) = create_depth_buffer( window, allocator );
```

Framebuffer The depth image simply becomes another attachment in the framebuffers. First, update the `create_swapchain_framebuffers` to accept another argument of type `VkImageView`:

```

void create_swapchain_framebuffers(
    lut::VulkanWindow const&,
    VkRenderPass,
    std::vector<lut::Framebuffer>&,
    VkImageView aDepthView
);

```

Then update the definition of the function, adding the passed-in depth image view the list of attachments:

```

void create_swapchain_framebuffers( lut::VulkanWindow const& aWindow, VkRenderPass ▽ 1
    ▽ aRenderPass, std::vector<lut::Framebuffer>& aFramebuffers, VkImageView ▽
    ▽ aDepthView )
{
    assert( aFramebuffers.empty() );

    for( std::size_t i = 0; i < aWindow.swapViews.size(); ++i )
    {
        VkImageView attachments[2] = {
            aWindow.swapViews[i],
            aDepthView // New!
        };

        VkFramebufferCreateInfo fbInfo{};
        fbInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
        fbInfo.flags = 0;
        fbInfo.renderPass = aRenderPass;
        fbInfo.attachmentCount = 2; // Updated!
        fbInfo.pAttachments = attachments;
        fbInfo.width = aWindow.swapchainExtent.width;
        fbInfo.height = aWindow.swapchainExtent.height;
        fbInfo.layers = 1;

        // ...
    }
}

```

Do not forget to update the attachmentCount of the VkFramebufferCreateInfo structure.

Finally, update the calls to create_swapchain_framebuffers. The function is called in two places. First during initialization and later when re-creating the swap chain.

Render pass To use the depth buffer, the render pass structure also needs to be updated. First, we need to declare that the render pass works with two attachments. Locate create_render_pass and extend the number of VkAttachmentDescriptions to two

```

VkAttachmentDescription attachments[2]{};
attachments[0].format = aWindow.swapchainFormat;
attachments[0].samples = VK_SAMPLE_COUNT_1_BIT;
attachments[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

attachments[1].format = cfg::kDepthFormat;
attachments[1].samples = VK_SAMPLE_COUNT_1_BIT;
attachments[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachments[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

```

The first attachment is unchanged (it is still the swap chain image). The second attachment refers to the depth buffer. The depth buffer should be cleared when the render pass begins (hence, loadOp is set to VK_ATTACHMENT_LOAD_OP_CLEAR). We do not care about the depth buffer's contents after this render pass finishes (and have only one renderpass!), and thus specify storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE. When using the depth buffer, it will be in the VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL layout. We do not care about the layout afterwards, so we set the final layout to this as well.

Do not forget to update the attachmentCount in the VkRenderpassCreateInfo structure below. It should be 2 now, instead of 1!

Next, we need to update the subpass to use this second attachment as the depth buffer. This is specified using the `pDepthStencilAttachment` member of the `VkSubpassDescription`. The member is set to point to a new `VkAttachmentReference` that refers to the depth attachment (attachment number 1):

```

VkAttachmentReference subpassAttachments[1]{};
subpassAttachments[0].attachment = 0; // this refers to attachments[0]
subpassAttachments[0].layout     = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

// New:
VkAttachmentReference depthAttachment{};
depthAttachment.attachment = 1; // this refers to attachments[1]
depthAttachment.layout     = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkSubpassDescription subpasses[1]{};
subpasses[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpasses[0].colorAttachmentCount = 1;
subpasses[0].pColorAttachments = subpassAttachments;
subpasses[0].pDepthStencilAttachment = &depthAttachment; // New!

```

Finally, we have to add another subpass dependency to make sure our accesses to the depth buffer synchronize properly between frames. Add another subpass dependency (keep the one that was defined in Exercise 1.3):

```

deps[1].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
deps[1].srcSubpass      = VK_SUBPASS_EXTERNAL;
deps[1].srcAccessMask   = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
deps[1].srcStageMask    = VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
deps[1].dstSubpass      = 0;
deps[1].dstAccessMask   = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT | ∇
    ▷ VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT;
deps[1].dstStageMask    = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT | ∇
    ▷ VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;

```

Don't forget to increase the size of the `deps` array to two, and make sure you update the `dependencyCount` in the `VkRenderPassCreateInfo`!

Graphics Pipeline When creating the graphics pipeline (`create_pipeline`), we must enable depth testing. Depth testing (and the stencil operations) are controlled by the [VkPipelineDepthStencilStateCreateInfo](#) structure, which we ignored in previous exercises. For now, we will focus on the options relating to depth testing, and leave the stencil tests be.

In the `create_pipeline` function, create and initialize a `VkPipelineDepthStencilStateCreateInfo` structure as follows:

```

VkPipelineDepthStencilStateCreateInfo depthInfo{};
depthInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthInfo.depthTestEnable = VK_TRUE;
depthInfo.depthWriteEnable = VK_TRUE;
depthInfo.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
depthInfo.minDepthBounds = 0.f;
depthInfo.maxDepthBounds = 1.f;

```

In essence, we enable both depth testing (which discards hidden fragments) and depth writes (which ensures that the depth buffer is updated during rendering). The `depthCompareOp` defines when a fragment passes the depth test. In our setup, the far plane is at 1.0 and the near plane is at 0.0 in the homogeneous coordinate space. A fragment with a lower depth value is therefore closer than one with a higher. Hence, we set the depth compare operation to `VK_COMPARE_OP_LESS_OR_EQUAL` – fragments with a lower (or equal) depth value will be considered to *pass* the depth test (and therefore be kept). Fragments that fail the depth test (higher depth value) are discarded instead.

Another choice might be `VK_COMPARE_OP_LESS`. The difference is minor in this case, as few fragments will have exactly equal depth values. Some multipass algorithms rely on rendering all geometry twice in exactly the same location. In this case `VK_COMPARE_OP_LESS_OR_EQUAL` is typically the right choice. (An example would be using a *pre-z* pass (no shading) to prime the depth buffer, and then performing a normal forward pass to do the shading.)



Make sure to point the `pDepthStencilState` member of the `VkGraphicsPipelineCreateInfo` to the newly created `depthInfo` structure!

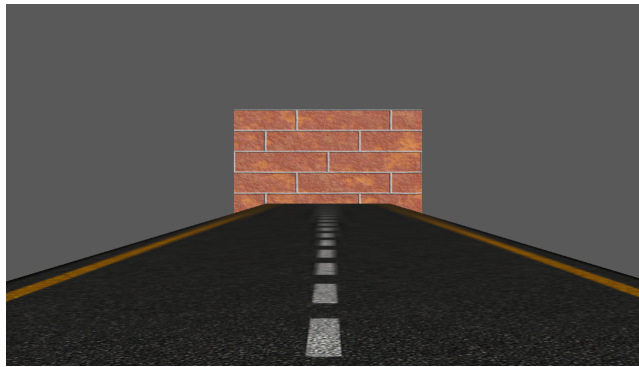


Figure 2: Two objects, with depth testing (Section 2). Compare to Figure 1, where the same scene was drawn without depth testing.

Drawing The final change takes place when recording commands for drawing. When starting the render pass with `vkCmdBeginRenderPass`, we must specify a clear value for the framebuffer.

Locate the definition of the `clearValues` array in `record_commands`. Extend it to a length of two to match the number of attachments that the render pass now uses. Set the clear value of the second to 1.0:

```
VkClearColor clearValues[2]{};
clearValues[0].color.float32[0] = 0.1f; // unchanged
clearValues[0].color.float32[1] = 0.1f;
clearValues[0].color.float32[2] = 0.1f;
clearValues[0].color.float32[3] = 1.f;

clearValues[1].depthStencil.depth = 1.f; // new!
```

Again, do not forget to update the `clearValueCount` in the `VkRenderPassBeginInfo` structure to two.

With this, we have created a depth buffer and enabled depth testing when rendering. Build and run the application. Compare your result to the image in Figure 2.

3 2nd pipeline & Blending

(New shaders — New Graphics Pipeline — Drawing)

We will switch out the brick texture (Figure 2) to the explosion sprite texture (see teaser on the first page). To draw the explosion sprite correctly, alpha blending is required. We could simply enable alpha blending in the main graphics pipeline and be done. However, to illustrate the use of multiple graphics pipelines, Exercise 1.4 introduces a second pipeline that has alpha blending enabled and leaves the original pipeline as is.

It is probably a good idea to separate alpha blended objects from fully opaque ones anyway. Correct transparency may require drawing alpha blended objects in the right order. Additionally, alpha blending is likely to incur a slight performance penalty. Drawing fully opaque objects separately with a pipeline that does not use alpha blending may therefore be more efficient. (The latter also applies to alpha masked objects vs. fully opaque ones.)



Begin by changing the `kSpriteTextures` to `explosion.png`.

Additional shaders The second pipeline will use a different fragment shader, `shaderTexAlpha.frag`. The current vertex shader, `shaderTex.vert` is simply reused.

Add the definitions to the `cfg` namespace:

```
constexpr char const* kAlphaVertShaderPath = SHADERDIR_ "shaderTex.vert.spv";
constexpr char const* kAlphaFragShaderPath = SHADERDIR_ "shaderTexAlpha.frag.spv";
```

(This should not replace the existing `kVertShaderPath` and `kFragShaderPath`!)

The new fragment shader is relatively simple. In the previous `shaderTex.frag`, we only use the RGB components of the texture, and explicitly set the alpha channel to 1. In the new shader, we simply use the full RGBA data returned from the texture and output this:


```

#version 450
1
2
layout( location = 0 ) in vec2 v2fTexCoord;
3
4
layout( set = 1, binding = 0 ) uniform sampler2D uTexColor;
5
6
layout( location = 0 ) out vec4 oColor;
7
8
void main()
9
10
{
    oColor = texture( uTexColor, v2fTexCoord ).rgba;
11
12
}

```

Graphics pipeline The new graphics pipeline uses the same inputs as the old one, meaning that we do not need to create a separate pipeline layout. We can focus on just creating the additional graphics pipeline.

Declare a new function, `create_alpha_pipeline` (naming is difficult; feel free to come up with a better name):

```

lut::Pipeline create_alpha_pipeline( lut::VulkanWindow const&, VkRenderPass, ▽
    ▸ VkPipelineLayout );
1

```

The definition of the function is largely identical to the current `create_pipeline` function. For now, just create a copy of the `create_pipeline` function definition and rename it to `create_alpha_pipeline`.

There are two changes to the function. First, instead of loading the shaders from `kVertShaderPath` and `kFragShaderPath`, load the shaders from `kAlphaVertShaderPath` and `kAlphaFragShaderPath`:

```

// Load shader modules
1
lut::ShaderModule vert = lut::load_shader_module( aWin, cfg::kAlphaVertShaderPath );
2
lut::ShaderModule frag = lut::load_shader_module( aWin, cfg::kAlphaFragShaderPath );
3

```

Next, update the `VkPipelineColorBlendAttachmentState` to enable blending:

```

VkPipelineColorBlendAttachmentState blendStates[1]{};
1
blendStates[0].blendEnable = VK_TRUE; // New! Used to be VK_FALSE.
2
blendStates[0].colorBlendOp = VK_BLEND_OP_ADD; // New!
3
blendStates[0].srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA; // New!
4
blendStates[0].dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA; // New!
5
blendStates[0].colorWriteMask = VK_COLOR_COMPONENT_R_BIT | ▽
6
    ▸ VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;

```

The code enables blending and configures “normal” alpha blending to take place.

Finally, make sure to create the new pipeline in `main()`:

```

lut::Pipeline alphaPipe = create_alpha_pipeline( window, renderPass.handle, ▽
    ▸ pipeLayout.handle );
1

```

The pipeline also needs to be recreated whenever the swap chain is recreated, so add the corresponding call to the swap chain recreation code!

Drawing To draw with the newly created graphics pipeline, we must bind it with the `vkCmdBindPipeline` command. By now, you can probably guess that this means adding another argument to `record_commands`. Update `record_commands` to accept an additional `VkPipeline` argument called e.g., `aAlphaPipeline`. Update the call to `record_commands` to pass the previously created `alphaPipe.handle` to this new argument.

We want to use the new pipeline with the second object (Section 1) only. In the `record_commands` function definitions, add a call to `vkCmdBindPipeline` just before issuing the draw calls for the second object:

```

// ...
1
vkCmdDraw( aCmdBuff, aVertexCount, 1, 0, 0 ); // initial object.
2
3
// New:
4
vkCmdBindPipeline( aCmdBuff, VK_PIPELINE_BIND_POINT_GRAPHICS, aAlphaPipe );
5
6
// Following calls were added for the second object
7
vkCmdBindDescriptorSets( aCmdBuff, VK_PIPELINE_BIND_POINT_GRAPHICS, aGraphicsLayout, ▽
    ▸ 1, 1, &aSpriteObjDescriptors, 0, nullptr );

```

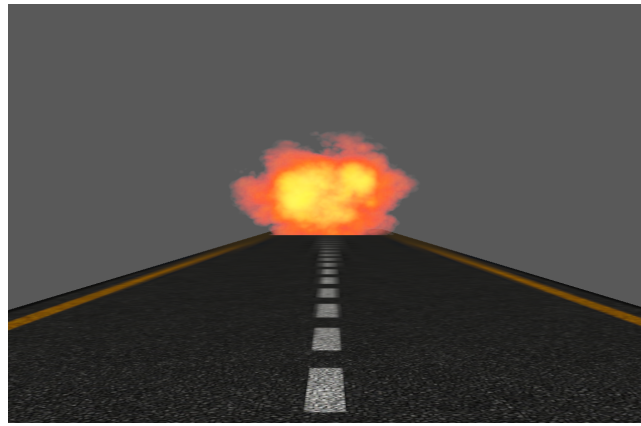


Figure 3: Alpha-blended “explosion” with a second pipeline.

```

VkBuffer spriteBuffers[2] = { aSpritePosBuffer, aSpriteTexBuffer };
VkDeviceSize spriteOffsets[2]{};

vkCmdBindVertexBuffers( aCmdBuff, 0, 2, spriteBuffers, spriteOffsets );

vkCmdDraw( aCmdBuff, aSpriteVertexCount, 1, 0, 0 );

```

Build and run the program. Verify that your output matches Figure 3.

4 User Camera

Last, we will add a user-controllable camera to the application. All infrastructure is in place, so this does not involve any new Vulkan code. Nevertheless, having a working camera is extremely useful for debugging.

In past COMP5822M iterations, there have been some very very janky camera implementations. These have made the life of everybody involved much more difficult than necessary. This camera implementation certainly isn’t perfect either, but it should work well enough to (for the most part) not translate the user to NaN randomly.



We will aim for first-person like camera that is controlled with the mouse and the standard (these days) WSAD control scheme. In addition, the keys E and Q are used to move up and down. Additionally, we will increase movement speed when shift is pressed, and reduce it when control is pressed. The camera’s movement speed should be independent of the frame rate (for example, we should not move faster just because the application is running at a higher frame rate). The camera controls are only active after the users clicks the right mouse button. Clicking the mouse button again deactivates the camera controls.

The goal is to make the camera useful for debugging. As such, we want to be able to navigate the scene quickly (the shift key will help here) but also have some fine control to inspect results up close (control key mode).

Our application already includes a camera transform. We just need to update it based on user input. The camera transform C is defined as the transform that takes us from world space to camera space. This essentially places the world in camera space. However, intuitively, it is often easier to place the camera in world space, which means that we model the camera-to-world transform (C^{-1}).

Preparations We use the C++ standard library to keep track of time. The standard header `<chrono>` provides functions related to this. If you use the Exercise 5 template, it has already been included; if you continued your work from Exercise 4, you will need to add the corresponding `#include <chrono>` line yourself.

To save us some typing, we’ll create a few aliases. Find the beginning of the anonymous namespace in `main.cpp` and add the following:

```

using Clock_ = std::chrono::steady_clock;
using Secondsf_ = std::chrono::duration<float, std::ratio<1>>;

```

First, this defines a type alias `Clock_` to map to `std::chrono::steady_clock`. We have a few different choices for the clock. The `steady_clock` represents a monotonic clock, that is, a clock that will never decrease

(go backward) in time.

Why would a clock run backwards? This can happen under some circumstances. For example, the system clock might get corrected by [NTP](#) or a similar mechanism. The user might change the clock (or the timezone might change). While these events are likely rare, using a clock that is guaranteed to be monotonic is still a good idea.



C++ defines several clocks, including `high_resolution_clock` and `system_clock`. However, neither of the two guarantee monotonicity. Traditionally, `high_resolution_clock` has also been a reasonable choice, as it has provided a decent resolution, even in early implementations. The quality of `system_clock` has varied quite a bit, and especially early versions had a poor resolution. Currently, on Windows and when using a recent VisualStudio, the `steady_clock` and `high_resolution_clock` are the same (the latter is just an alias to the former). In contrast, the `system_clock` has a separate implementation. All three clocks appear to have the same cost and precision. In Linux (with a current GCC/libstdc++), the `system_clock` and the `high_resolution_clock` are the same. All three clocks also have the same cost and resolution. However, clock resolution on Windows seems overall worse than in Linux. Under Linux, the clocks and the underlying `clock_gettime` system call all achieve resolution and performance similar to the `x86 RDTSC/RDTSCP` instructions. On an example system, the cost and resolution of all those clocks is around 20ns or 75 cycles in Linux. On the same system, Windows' clocks, including `QueryPerformanceCounter`, instead have a minimal resolution of 100ns (cost is similar as in Linux). To get any better, one has to use specialized instructions like `RDTSCP` on x86.



The `Secondsf_` type represents a duration (time span), where one unit corresponds to one second. It uses a `float` to store the duration, meaning that fractional seconds are possible. The choice for floating point times in seconds makes it easy to intuitively understand various quantities (e.g., speed is simply units per second). We only use the duration type for short time deltas (around a frame), so floats provide sufficient precision.

If you store long durations such as game time, you should use a `double`. [John Carmack](#) agrees [warning: Twitter/X]. Longer discussion available [here](#). See “Time deltas fit in a float” before you send me angry emails (again).



We will then define a few constants in the `cfg` namespace. These relate to the camera configuration and are mostly empirically determined values. For scenes and scene setups, you might want to use different settings:

```
// Camera settings.
//
// These are determined empirically (i.e., by testing and picking something that felt OK).
//
// General rule: for debugging, you want to be able to move around quickly in the scene (but slow down if
// necessary). The exact settings here depend on the scene scale and similar settings.
constexpr float kCameraBaseSpeed = 1.7f; // units/second
constexpr float kCameraFastMult = 5.f; // speed multiplier
constexpr float kCameraSlowMult = 0.05f; // speed multiplier

constexpr float kCameraMouseSensitivity = 0.01f; // radians per pixel
```

The application receives user input via GLFW's callbacks. One such callback was introduced in Exercise 1.3 to receive key presses: `glfw_callback_key_press`. We will add two more:

```
void glfw_callback_button( GLFWwindow*, int, int, int );
void glfw_callback_motion( GLFWwindow*, double, double );
```

The first callback receives mouse button events. The second one receives the mouse position whenever the mouse is moved.

We will define a few helper types to keep track of the current state:

```
enum class EInputState
{
    forward,
    backward,
    strafeLeft,
    strafeRight,
    levitate,
    sink,
```

```

    fast,
    slow,
    mousing,
    max
};

struct UserState
{
    bool inputMap[std::size_t(EInputState::max)] = {};

    float mouseX = 0.f, mouseY = 0.f;
    float previousX = 0.f, previousY = 0.f;

    bool wasMousing = false;

    glm::mat4 camera2world = glm::identity<glm::mat4>();
};

```

Also declare a `update_user_state` function to update the state based on the elapsed time:

```
void update_user_state( UserState&, float aElapsedTime );
```

Receiving Input We will first implement the GLFW callbacks. We need a way to communicate data out of the callback functions, back to the main program. GLFW provides a mechanism with a user-pointer for this purpose. We can associate an arbitrary pointer with a `GLFWwindow` using the [glfwSetWindowUserPointer](#) function. Once set, we can retrieve the pointer again using [glfwGetWindowUserPointer](#). GLFW never inspects this pointer internally, so we're free to use it to store whatever. (This is a fairly common pattern in C APIs that expose callbacks.)

We begin by setting up the user pointer. It will point to a `UserState` instance. In `main()`, just after creating the window with `make_vulkan_window`, declare a new `UserState` instance and then register a pointer to it with the window:

```

// Configure the GLFW window
UserState state{};

glfwSetWindowUserPointer( window.window, &state );

```

We will also register the remaining callbacks with GLFW:

```

glfwSetKeyCallback( window.window, &glfw_callback_key_press ); // This was added in Ex 1.3
glfwSetMouseButtonCallback( window.window, &glfw_callback_button );
glfwSetCursorPosCallback( window.window, &glfw_callback_motion );

```

Next, we'll implement the callbacks. The key press callback notifies us whenever the state of a key changes. Currently it handles pressing *Escape*, which terminates the application. Below this check, we will handle the keys that we want to control the camera and simply toggle the relevant states in `UserState::inputMap`. The `glfwGetWindowUserPointer` returns a pointer to the `UserState` instance; it is untyped (`void*`), so we will have to cast it to the proper type first:

```

auto state = static_cast<UserState*>(glfwGetWindowUserPointer( aWindow ));
assert( state );

bool const isReleased = (GLFW_RELEASE == aAction);

switch( aKey )
{
    case GLFW_KEY_W:
        state->inputMap[std::size_t(EInputState::forward)] = !isReleased;
        break;
    case GLFW_KEY_S:
        state->inputMap[std::size_t(EInputState::backward)] = !isReleased;
        break;
    case GLFW_KEY_A:
        state->inputMap[std::size_t(EInputState::strafeLeft)] = !isReleased;
        break;
}

```

```

    case GLFW_KEY_D:
        state->inputMap[std::size_t(EInputState::strafeRight)] = !isReleased;
        break;
    case GLFW_KEY_E:
        state->inputMap[std::size_t(EInputState::levitate)] = !isReleased;
        break;
    case GLFW_KEY_Q:
        state->inputMap[std::size_t(EInputState::sink)] = !isReleased;
        break;

    case GLFW_KEY_LEFT_SHIFT: [[fallthrough]];
    case GLFW_KEY_RIGHT_SHIFT:
        state->inputMap[std::size_t(EInputState::fast)] = !isReleased;
        break;

    case GLFW_KEY_LEFT_CONTROL: [[fallthrough]];
    case GLFW_KEY_RIGHT_CONTROL:
        state->inputMap[std::size_t(EInputState::slow)] = !isReleased;
        break;

    default:
        ;
}

```

The mouse button callback is fairly similar. It checks for presses of the right mouse button. When pressed, it toggles the `EInputState::mousing` state to indicate that mouse-look is now active. To make things a bit more user-friendly, we will hide the cursor in this mode (mouse movements are linked directly to the camera). GLFW provides [glfwSetInputMode](#) for this purpose. It also has the side effect that the cursor is fixed to our window and cannot leave it (letting the user exit navigation mode to get back control of the mouse again is therefore an important feature).

```

void glfw_callback_button( GLFWwindow* aWin, int aBut, int aAct, int )
{
    auto state = static_cast<UserState*>(glfwGetWindowUserPointer( aWin ));
    assert( state );

    if( GLFW_MOUSE_BUTTON_RIGHT == aBut && GLFW_PRESS == aAct )
    {
        auto& flag = state->inputMap[std::size_t(EInputState::mousing)];

        flag = !flag;
        if( flag )
            glfwSetInputMode( aWin, GLFW_CURSOR, GLFW_CURSOR_DISABLED );
        else
            glfwSetInputMode( aWin, GLFW_CURSOR, GLFW_CURSOR_NORMAL );
    }
}

```

The final callback reports the position of the mouse cursor whenever the mouse is moved. Our implementation just updates the relevant variables in the `UserState` structure:

```

void glfw_callback_motion( GLFWwindow* aWin, double aX, double aY )
{
    auto state = static_cast<UserState*>(glfwGetWindowUserPointer( aWin ));
    assert( state );

    state->mouseX = float(aX);
    state->mouseY = float(aY);
}

```

Updating camera state Each frame, in the main loop, we will update the camera's transform based on the current input state and the elapsed time.

Camera movement consists of a rotation R and a translation T . The rotation R is derived from the distance the mouse cursor has moved along each axis. We just consider the total movement since the last frame. In this case, we do not consider the elapsed time, and instead let the rotation depend only on the physical distance

that the mouse has moved. The translation is instead based on what keys are pressed and the distance of is determined by a movement speed and the elapsed time.

The updated camera transformation, C'^{-1} , becomes $C'^{-1} = C^{-1}RT$, where C^{-1} is the previous frame's camera to world transformation.

Implement this in the `update_user_state` method:

```

void update_user_state( UserState& aState, float aElapsedTime )      1
{                                                                    2
    auto& cam = aState.camera2world;                                3

    if( aState.inputMap[std::size_t(EInputState::mousing)] )        4
    {                                                                    5
        // Only update the rotation on the second frame of mouse navigation. This ensures that the previousX  6
        // and Y variables are initialized to sensible values.        7
        if( aState.wasMousing )                                       8
        {                                                              9
            auto const sens = cfg::kCameraMouseSensitivity;          10
            auto const dx = sens * (aState.mouseX-aState.previousX); 11
            auto const dy = sens * (aState.mouseY-aState.previousY); 12
                                                                    13
            cam = cam * glm::rotate( -dy, glm::vec3( 1.f, 0.f, 0.f ) ); 14
            cam = cam * glm::rotate( -dx, glm::vec3( 0.f, 1.f, 0.f ) ); 15
                                                                    16
        }                                                            17
                                                                    18
        aState.previousX = aState.mouseX;                            19
        aState.previousY = aState.mouseY;                            20
        aState.wasMousing = true;                                     21
    }                                                                    22
    else                                                                23
    {                                                                    24
        aState.wasMousing = false;                                    25
    }                                                                    26
                                                                    27
    auto const move = aElapsedTime * cfg::kCameraBaseSpeed *         28
        (aState.inputMap[std::size_t(EInputState::fast)] ? cfg::kCameraFastMult:1.f) * 29
        (aState.inputMap[std::size_t(EInputState::slow)] ? cfg::kCameraSlowMult:1.f) 30
    ;                                                                    31
                                                                    32
    if( aState.inputMap[std::size_t(EInputState::forward)] )         33
        cam = cam * glm::translate( glm::vec3( 0.f, 0.f, -move ) ); 34
    if( aState.inputMap[std::size_t(EInputState::backward)] )         35
        cam = cam * glm::translate( glm::vec3( 0.f, 0.f, +move ) ); 36
                                                                    37
    if( aState.inputMap[std::size_t(EInputState::strafeLeft)] )       38
        cam = cam * glm::translate( glm::vec3( -move, 0.f, 0.f ) ); 39
    if( aState.inputMap[std::size_t(EInputState::strafeRight)] )      40
        cam = cam * glm::translate( glm::vec3( +move, 0.f, 0.f ) ); 41
                                                                    42
    if( aState.inputMap[std::size_t(EInputState::levitate)] )         43
        cam = cam * glm::translate( glm::vec3( 0.f, +move, 0.f ) ); 44
    if( aState.inputMap[std::size_t(EInputState::sink)] )             45
        cam = cam * glm::translate( glm::vec3( 0.f, -move, 0.f ) ); 46
    }                                                                    47
}

```

The `update_user_state` needs to be called each frame in the main loop. One of its arguments is the elapsed time since the last update, so we have to first compute it.

Just before entering the main loop, record the current time:

```

auto previousClock = Clock_::now();                                  1

```

This mainly just declares the `previousClock` variable, but it is useful to initialize it to a reasonable value for the first iteration of the loop.

In the main loop, *after* waiting for the command buffer to become available with a fence, but before updating the scene uniform data (`update_scene_uniforms`), compute the elapsed time and call `update_user_state`:

```

// Update state
auto const now = Clock_::now();
auto const dt = std::chrono::duration_cast<Secondsf_>(now-previousClock).count();
previousClock = now;

update_user_state( state, dt );

```

Finally, we need to pass the camera transform to our shaders. The `update_scene_uniforms` already updates the uniform buffers each frame, so we can just pass the `UserState` to it and have it use the updated camera transform.

Update `update_scene_uniforms`'s declaration to take an additional `UserState` argument (use `const&` – we do not intend to change the user state from `update_scene_uniforms`). In the main loop, update the invocation of `update_scene_uniforms` to pass the `state` object as the last argument. Finally, update the implementation of `update_scene_uniforms` to derive the camera transform from the `camera2world` member of the `UserState`:

```

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

// Now:
aSceneUniforms.camera = glm::inverse( aState.camera2world );

```

(Here I named the `UserState` argument to `update_scene_uniforms` `aState`.)

You should now be able to navigate the scene with WSAD+EQ, and look around with the mouse after activating mouse navigation with a right click. Make sure this works as expected.

Wrapping Up

Exercise 1.5 introduced a second object and showed the need for depth buffering. Additionally, enabling alpha blending demonstrated the utility of multiple graphics pipelines.

While Exercise 1.5 (and Exercise 1.4) sticks to simple geometry (a total of 4 triangles), the fundamentals shown here scale to much larger objects. Instead of passing buffers with two triangles, we could as easily pass buffers containing data for 100'000+ vertices and thereby rendering quite complex objects.

Exercise 1.5's handling of multiple objects is not quite ideal yet. Manually adding arguments to a function for each additional object is unlikely to scale – in fact, `record_commands` is quite unwieldy already. Some sort of scene data structure listing and managing all objects would be required. (Exercise 1.5 avoids this on purpose, though, as designing a good data structure for this is an entire exercise by itself.) Nevertheless, you should now have seen the fundamental components necessary for creating a fully fledged Vulkan renderer. Subsequent steps are mostly more of the same – more textures, more buffers, more shaders, more framebuffers, and so on.

As always, make sure there are no (unexpected) Vulkan validation errors when you run the program. Use the Vulkan configurator tool (Exercise 1.3) to check for synchronization errors as well.

Optional experiments





Reverse-Z [advanced!] Reverse-Z is a technique that better utilizes the 32-bit float depth buffer that Exercise 1.5 uses by default. The name stems from the fact that Reverse-Z reverses the depth buffer range, i.e., the near plane ends up at 1.0 and the far plane at 0.0 in clip space. With Reverse-Z rendering, it is possible to push the (world space) far plane much further out, enabling rendering of larger scenes. In fact, it is even possible to place the far plane at infinity with minimal loss of precision in depth close to the camera.

Reverse-Z requires an adjusted projection matrix, and some changes to the depth buffer setup. You can find detailed information about the Reverse-Z technique via the following links:

- <https://nlguillemot.wordpress.com/2016/12/07/reversed-z-in-opengl/> (focus on OpenGL, has additional links.)
- <https://developer.nvidia.com/content/depth-precision-visualized> (NVIDIA's take on Reverse-Z. Non-interactive graphs visualizing the precision.)

You can easily find additional information by searching the web.

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.