coursework one - report

Yanni Ma

March 2024

1 Vulkan infrastructure

The details of the Vulkan implementation are as follows:
Vulkan device name: NVIDIA GeForce RTX 4050 Laptop GPU (1.3.271)
Device's Vulkan versions: 1.3.271
Loader's Vulkan versions: 1.3.268 (variant 0)
Enabling layer: VK_LAYER_KHRONOS_validation
Enabling instance extension: VK_KHR_surface
Enabling instance extension: VK_KHR_win32_surface
Enabling instance extension: VK_EXT_debug_utils
Enabling device extension: VK_KHR_swapchain
The color format: VK_FORMAT_B8G8R8A8_SRGB
The number of images: 3

2 3D Scene and Navigation

Traverse the mesh and save the position, color and texture coordinates in three std::vector<float>. If there are no texture coordinates, store (0.0f, 0.0f). Then for each mesh, create an objMesh struct. The objMesh struct consists of lut::Buffer positions, lut::Buffer texcoords, lut::Buffer colors, lut::Buffer indices, std::uint32_t indexCount, lut::Image meshImage, lut::ImageView meshImageView, VkDescriptorSet matDescriptor composition. Then I use a std::vector<objMesh> to save all the data.

```
1 std::vector<objMesh> create_mesh(labutils::VulkanContext const&
2 aWindow, lut::Allocator const& aAllocator, SimpleModel const&
3 aModel, VkCommandPool& aLoadCmdPool, VkDescriptorPool& aDesPool,
4 VkSampler& aSampler, VkDescriptorSetLayout& aLayout)
5 {
6         std::vector<objMesh> result;
7         for (auto mesh : aModel.meshes)
```

```
        {
            std::vector<float> position;
            std::vector<float> texture;
            std::vector<float> color;
            std::vector<uint32_t> indexData;
            std::uint32_t index = 0;
            ......
        }
        ......
}
```

About the costs of required Vulkan resources. I need GPU-only buffers for positions, textures, colors, indices for each mesh. Staging buffers for CPU-to-GPU data transfers for each mesh. And creating descriptor sets for textured meshes.

About the costs of required Vulkan operations. I did these operations: buffer copy, descriptor set updates, buffer barriers, command buffer recording, submission and waiting.

Load time: load mesh data, create buffer, transfer data, create descriptor set, command buffer recording, submission and waiting.

Each frame: place buffer barriers to ensure correct synchronization, update descriptor sets when necessary, render and present to the screen.

The results are as follows:

Figure 1: render result

3 Anisotropic filtering

I added a few lines of code to the create_default_sampler function in vkutil.cpp:

```
VkPhysicalDeviceProperties props{};
vkGetPhysicalDeviceProperties(aContext.physicalDevice, &props);
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = props.limits.maxSamplerAnisotropy;
```

The results are as follows:

Figure 2: Turn off Anisotropic filtering

Figure 3: Turn on Anisotropic filtering

I output the value of props.limits.maxSamplerAnisotropy to get the max sampler anisotropy is 16.0. By turning on anisotropic filtering, the folds of the lion's head show more details and more realistic.

4 Visualizing fragment properties

In the fragment shader, I get the mipmap level through the textureQueryLod function. Then I visualized the mipmap level by using the mix function and the smoothstep function. The result is as follows:

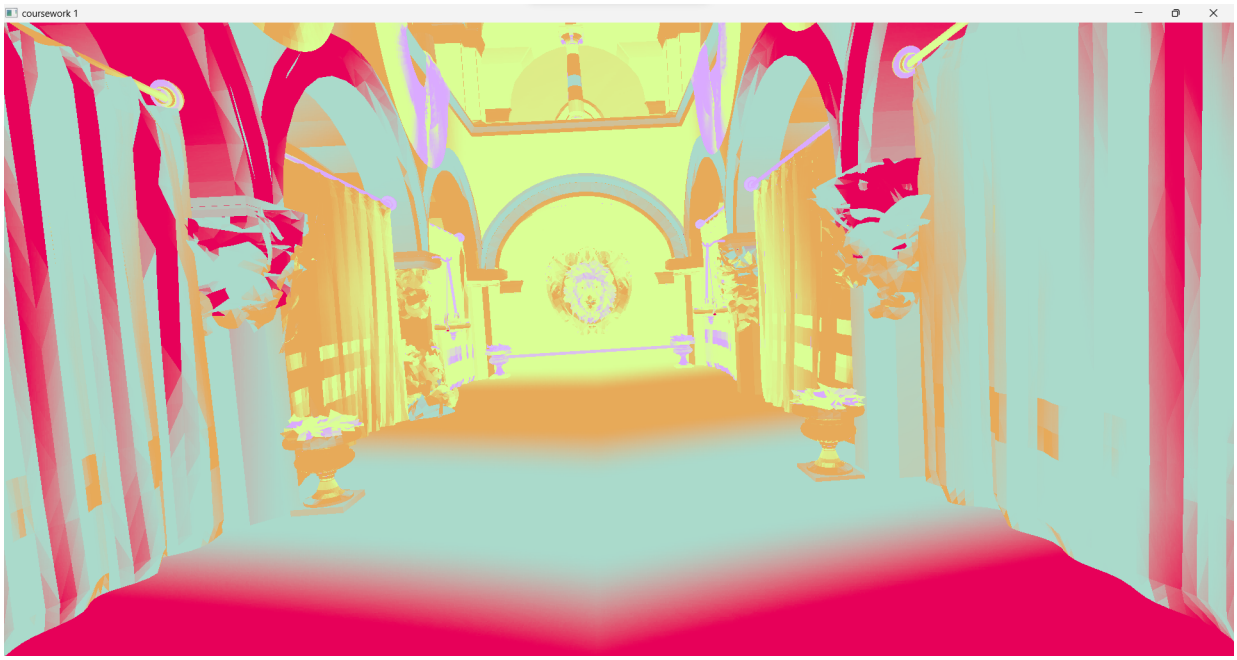

Figure 4: mipmap

I have gl_FragCoord.z as the depth value. To visualize the depth value, I scaled it exponentially. The result is as follows:

Figure 5: depth

To visualize the partial derivatives of the per-fragment depth, I multiply gl_FragCoord.z by 100. Then I use the dFdx and dFdy functions to calculate the partial derivatives. Finally, the length, mix, and smoothstep functions are used for visualization. The result is as follows:

Figure 6: the partial derivatives of the per-fragment depth

All mipmap levels are used. By adjusting the window size I found that the smaller the window, the higher the mipmap level. The larger the window, the smaller the mipmap level. A positive bias may cause a higher mipmap level, and a negative bias may cause a lower mipmap level.

5 Visualizing overdraw and overshading

overdraw: Turn off depth test and depth write, then turn on additive blending, and finally output a dark green color in the fragment shader. The result is as follows:

Figure 7: overdraw

overshading: Turn on depth test and depth write, then turn on additive blending, and finally output a dark green color in the fragment shader. The result is as follows:

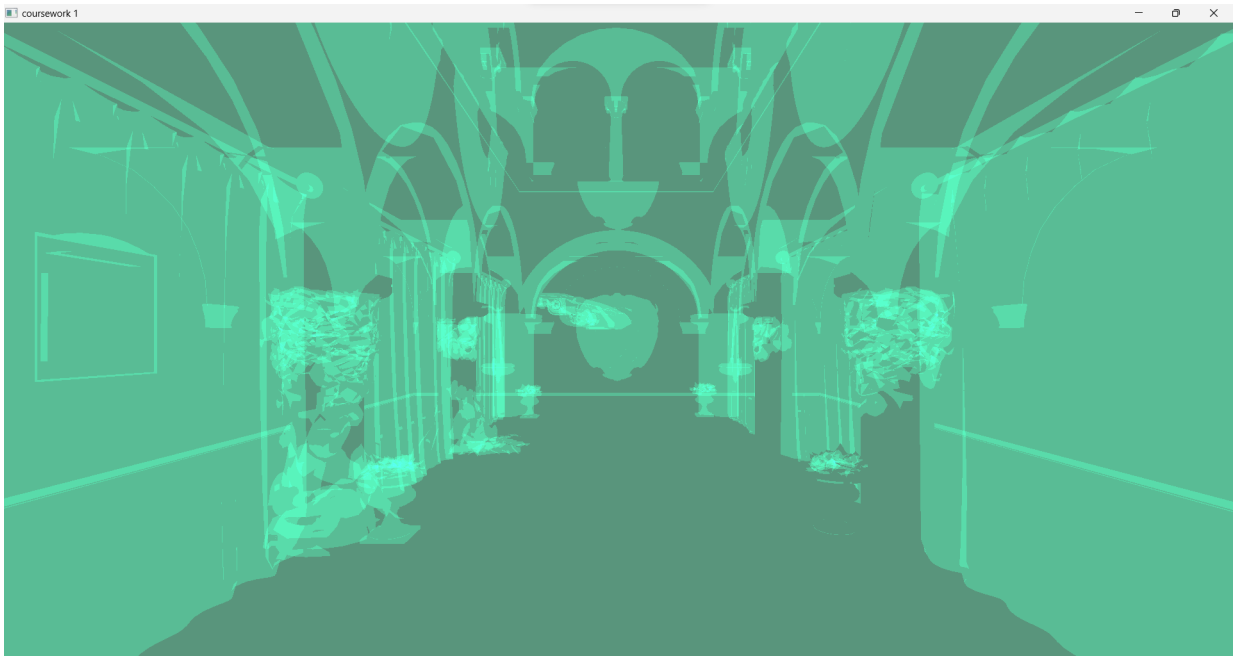

Figure 8: overshading

When the scene moves, there are different visual effects. At certain angles, it can be observed that overdrawing and overshading on one side are more obvious.

I predict there will be differences. Because at certain angles, there may be more objects overlapping each other, causing overdrawing and overshading to be more noticeable.

Reduce the baseline overdraw: spatial culling, frustum clipping and LOD control.

Reduce the baseline overshading: optimize shadow casting, control lighting parameters and pptimize materials.

6 Visualizing Mesh Density

I represented the density by adding a geometry shader and then calculating the area in it. The result is as follows:

Figure 9: Mesh Density

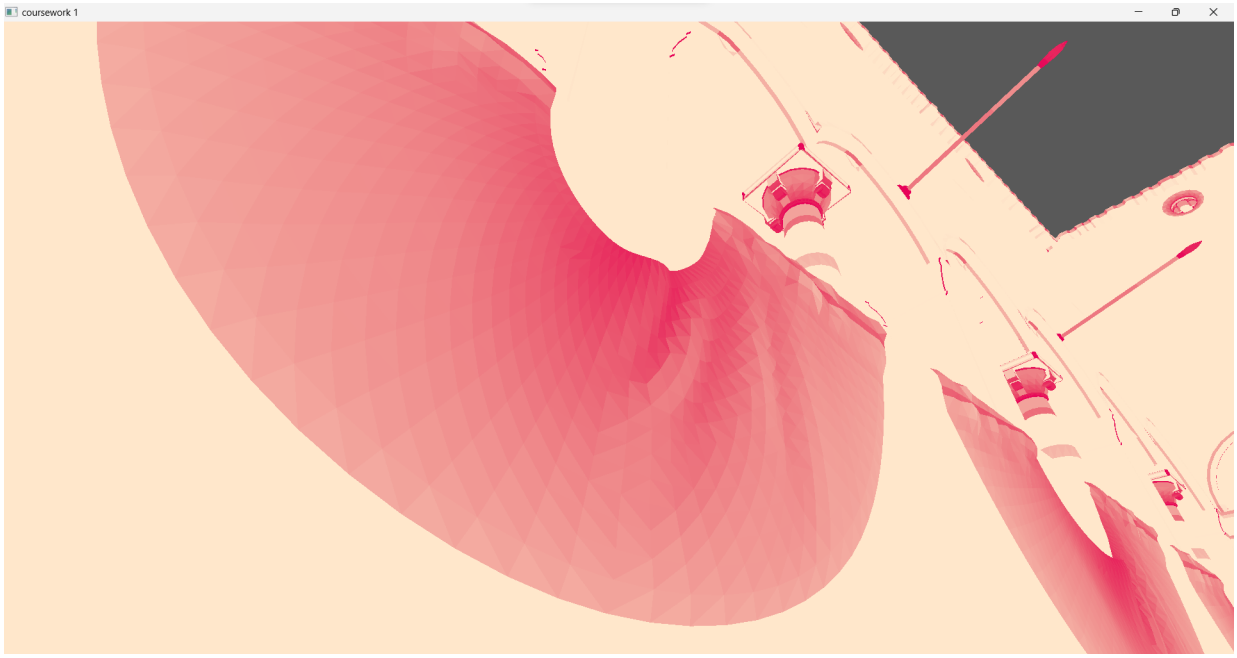

Figure 10: Mesh Density

The results show that the mesh density around complex geometric structures is higher. Distant or less important areas have lower mesh density.

Suggestion for scenes: Reduce mesh density in simple geometric areas.

Suggestion for rendering methods: Dynamically adjust mesh density based on distance from camera.