Report

ID: 201598922

1.1 Vulkan Infrastructure

According to my printed information about selected device and enabled extensions and features, my reported information as follows:

Vulkan device name: NVIDIA GeForce RTX 2060 with Max-Q Design
Vulkan version: 1.3.224
Vulkan Loader version: 1
Enabled Vulkan Instance extensions:
VK_KHR_surface
VK_KHR_win32_surface
VK_EXT_debug_utils

where VK_KHR_surface is necessary to create the Vulkan window, and VK_KHR_win32_surface which is necessary for me to create the window on Windows11 system. and VK_EXT_debug_utils is necessary to enable the debug utils from Vulkan SDK, and output validation error in the terminal while running the application. But it only activate in the debug mode.

Enabled Vulkan Device extensions:
VK_KHR_swapchain
VK_KHR_pipeline_library
VK_EXT_graphics_pipeline_library
VK_KHR_shader_non_semantic_info

where VK_KHR_swapchain is necessary for creating and using swap chain. VK_KHR_pipeline_library and VK_EXT_graphics_pipeline_library are aimed to improve the performance of graphics pieleine. VK_KHR_shader_non_semantic_info is used for debugging in the shader and print necessary information in the terminal.

Enabled Queue Families: Graphics Queues, Compute Queues
Where Graphics Queues is for graphics rendering. and Compute queues are used for accelerating computing task in the compute shader. which I used in the task 1.5.

Enabled Vulkan Physical Device Features: samplerAnisotropy
And the feature of samplerAnisotropy is used in task 1.3 to support anisotropy sampling.

Swap Chain color formats:
depth image format = VK_FORMAT_D32_SFLOAT
texture image format = VK_FORMAT_R8G8B8A8_SRGB

Swap Chain Image (View) size = 3

one is for depth, and the other two for rendering on the screen. The color formats type comes from previous exercises from 1.4 to 1.5, and I found it works.

1.2 3D Scene and Navigation

1.2.1 Strategy and Motivation of rendering meshes:

I found those of meshes has been divide into two groups:

1. Vertices with textures, which could be presented as,

    position:vec3, texCoord:vec2.

2. Vertices without textures, which could be presented as,

    position:vec3, color:vec3.

For rendering meshes, I believe the best practice render them as two different groups, and using two sets of shaders, pipeline layout, pipeline, and command buffers... But I didn't take this method, because I don't much familiar with Vulkan yet, and in the following tasks of 1.4 and 1.5, I need to render the meshes with solid colors. What's more, to make my approach has more versatility to support rendering other models. So I designed each mesh vertex as follows:

```
vec3 position;
vec3 colors;
vec2 texCoord;
```

Where colors are diffuse color in the .mtl file.This approach means I can only create one pipeline, pipeline layout... and I could reuse the pipeline in the 1.3 and 1.5. And it is easier to implement. In detail, I filled untextured meshes with zero. and all color come from material.

And there are some meshes without materials or colors, such as planes in the scene. In practice, Vulkan will report an error in the validation layer whiling filling texture samplers. My solution is to create a "default pixel" all this situation. Here I define this pixel as {255, 255, 255, 0}. (I specify the color of alpha channel, because all material color is albedo.)

1.2.2 Brief Process from Raw Mesh to Screen

Here is the brief process from raw data of meshes and materials to rendering on the screen:

1. Mesh Creation: Construct each mesh with attributed vertices form, and group them into a vector container for next steps.
2. Vertex Buffer Creation: Build up vertex buffer from meshes as separated parts including positions, colors and uvs, complete memory allocation with from CPU to GPU.
3. Read from textures to build up images, imageViews and sampler, using default sampler to fill the untextured fields.

> Steps 1 - 3 could be seen as resources initiations, there are two kinds of resources in Vulkan, buffer(view) and ImageViews. So, they are in loading time.

4. Create a graphics pipeline and relevant Vulkan objects descriptor sets, descriptor sets layout, pipeline layout, render passes... At same time, writing vertex/fragment shaders and other stuff.

> Steps 4 describes how to use the resources and what operations should be done in the runtime,

5. Command Execution: do command buffer and render pass execution, whiling taking the vertex buffer in.
6. Submit to queue and then present, using fences, barriers, to sync processes.

> Steps 5 - 6 happens each frame, update rendering results each frame. If there will be something interrupted, like resize the window, or move camera position, the pipeline will be recreated and updated. For improving the performance, I use the `pipeline cache` for the graphics pipeline (though it is introduced will be more useful If I switch multiple pipelines).

1.2.3 Simple Analysis for Extra Cost

Considering about costs. it fills the wasted data in textures in a vec2, which size is `2 * sizeof(float)` in the untextured vertices.

So, It will cost extra `2 * sizeof(float) * untextured_vertices.size()`.

And there is the value of `untextured_vertices.size()` is 102798. I work on x64 plateform, thus the sizeof(float) is 8 bytes. Meanwhile, using the single `white pixel` for filling, which size is 4 * sizeof(uint8_t), in 4 bytes. So, in total, I need to use extra data in CPU as follows:

extra cost in CPU = 102798 * 8 + 4 = 822388 bytes = 804kb.

As a result, thought it waste much meomery, but it is acceptable for me.

1.2.4 Render Result

1.3 Anisotropic filtering

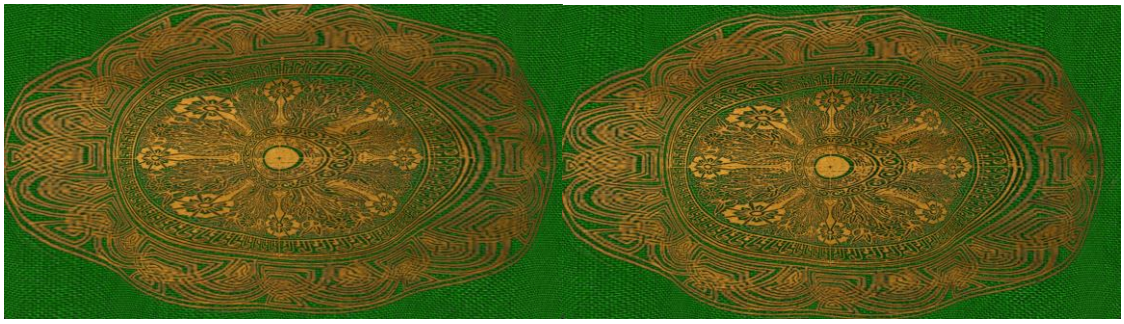
To enable anisotropic filetering, I do these changes in code:

In vulkan_window.cpp, add feature:

```
    VkPhysicalDeviceFeatures deviceFeatures{};
    deviceFeatures.samplerAnisotropy = VK_TRUE;
```

In vkutil.cpp, enbale anisotropy while creating sampler:

```
if(_enableAF)
samplerInfo.anisotropyEnable = VK_TRUE;
    samplerInfo.maxAnisotropy = 16.0f;
```

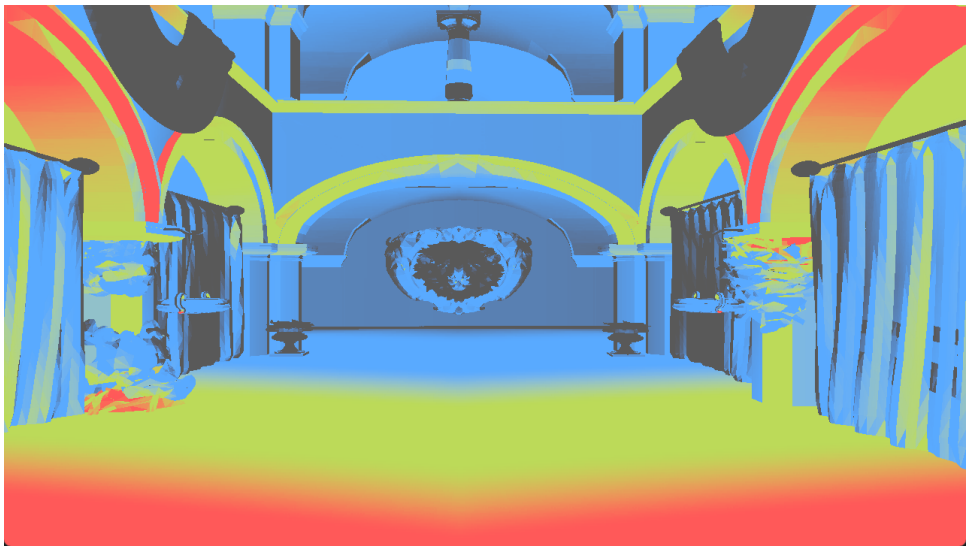


The left picture is the result without anisotropic filtering, and the right picture is the result with anisotropic filtering, which set the MaxSamplerAnisotropy is 16.0f. Because the cloth is not flat, so if without anisotropic filtering, at the slanted creases of fabric, there can be noticeable and unrealistic changes, even some jagged edges, but with anisotropic filtering enabled, the slanted surfaces appear smoother and the fabric looks smoother, giving a more realistic feel.

1.4 Visualizing Mipmaps

In line 359 of main.cpp. I set up a #ifdef mipmap to switch different shaders and uncomment marcos in line 4 #define mipmap to render the result of mipmap. And in the mipmap fragment shader, I use the textureQueryLod to get the mipmap levels,

and using linear interpolation to mix colors to render the results. The main method of implementing this task refer exercise 1.3 and 1.4.
And here is the rendering result of mipmap visualization.



Where, the red part is nearest and then move the green（yellow）, next to blue, finally to grey.
I have two monitor. And one is 1920*1080p, one is 2560*1440, and when I move the window from 1080p to 2k monitor, the red part increased, which means the higer resolution mipmaps are used more. When I resize the window bigger, the red part also increse more, so, I think about some optimisation methods:

1. Dynamic mipmap level selection, or using mipmap to choose mesh LOD levels
2. Limit mipmap generation farer areas, all in a low level.

1.5 Visualizing Mesh Density

According to the requirement of this task, the definition of mesh denstiy is open, so, I define it as:
number_of_triangle / total_area_of_a_mesh.
And then I map them into (0,1) with some weights to make the rendering results more visiable.
To compute the number of triangles and total area of a mesh. I use a compute shader and compute pipeline, to make the rendering reuslt could be viewed in real-time rendering.
Due to using compute shader, this approach could be used in many applications could compute shader, like OpenGl, DX11, DX12.
Here is the rendering result of my mesh density.

In my result, where the green means low density and red is the middle density, and while are the high density area.

However, there are serveal drawbacks of my implementaion:

1. It is viewpoint-independent now, that means, it not ignore the hidden faces. So, the reuslt may be so good, because it did not show out the accruate mesh rendering situation. I am suprivsed about the color both side are with high density.

Mesh density can be useful for developers to quickly understand the distribution of mesh density in a scene and optimize rendering performance.

Based on the results of this visualization method, modifications can be made to the provided scene, such as increasing mesh density in high-density areas to improve image quality, or reducing mesh density in low-density areas to improve rendering performance.