COMP5822M High Performance Graphics Coursework 1 Report

Simiao Wang 201702881

March 7, 2024

For the convenience of demonstration, change the '#define VERSION 0' to other number in the '#define' section of 'current_Version.hpp', then compile and run.

```
    #define DEFAULT 0
    #define MIPMAP_VISUALIZE_LINEAR 1
    #define MIPMAP_VISUALIZE_NEAREST 2
    #define FRAGMENT_DEPTH 3
    #define PARTIAL_DERIVATIVES_FRAGMENT_DEPTH 4
    #define ANISOTROPIC_FILTERING 5
    #define OVERDRAW_VISUALIZE 6
    #define OVERSHADING_VISUALIZE 7
    #define MESHDENSITY_VISUALIZE 8

    #define CURRENT_VERSION 0
```

The presetting to numbers here is because string comparison cannot be performed later in the #if statement.

1 Vulkan infrastructure

Vulkan device name: NVIDIA GeForce RTX 3060 Laptop GPU
Vulkan Loader's versions: 1.3.277 (variant 0)
Device's Vulkan versions: 1.3.277
Extensions:VK_KHR_surface, VK_KHR_win32_surface, VK_EXT_debug_utils, VK_KHR_swapchain
Swap chain images_Color format: VK_FORMAT_R8G8B8A8_SRGB
Swap chain images_Number of images: 3

2 3D Scene and Navigation

2.1 The way to handle different meshes and materials

If the mesh has textures, load their corresponding texture images and create texture image views.
If the mesh does not have textures, use the default sprite texture.
Create descriptor sets and allocate one for textured mesh. In the descriptor sets, bind information such as texture images and texture samplers to the corresponding descriptors.

```
if(mesh.textured)
{
    std::vector<float> tempPos;
    std::vector<float> tempTex;
    for (std::size_t i = 0; i < mesh.vertexCount; i++)
    {
        tempPos.push_back(Model.dataTextured.positions[mesh.vertexStartIndex + i].x);
        tempPos.push_back(Model.dataTextured.positions[mesh.vertexStartIndex + i].y);
        tempPos.push_back(Model.dataTextured.positions[mesh.vertexStartIndex + i].z);
        tempTex.push_back(Model.dataTextured.texcoords[mesh.vertexStartIndex + i].x);
        tempTex.push_back(Model.dataTextured.texcoords[mesh.vertexStartIndex + i].y);
    }
    {
        lut::CommandPool loadCmdPool = lut::create_command_pool(window,
            VK_COMMAND_POOL_CREATE_TRANSIENT_BIT);
        modelTex.push_back(lut::load_image_texture2d(Model.materials[mesh.materialIndex].diffuseText
            window, loadCmdPool.handle, allocator));
```

```
        modelView.push_back(lut::create_image_view_texture2d(window,
            modelTex.back().image, VK_FORMAT_R8G8B8A8_SRGB));
    }
    tempDescriptors = 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 = modelView.back().handle;
        textureInfo.sampler = defaultSampler.handle;
        desc[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
        desc[0].dstSet = tempDescriptors;
        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);
    }
    TexturedMesh tempMesh = create_model_mesh(window, allocator, tempPos, tempTex);
    modelMesh.push_back(std::move(tempMesh));
    modelDescriptors.push_back(std::move(tempDescriptors));
}
else
{
    std::vector<float> aircraftTempPos;
    std::vector<glm::vec3> aircraftColor;
    for (std::size_t i = 0; i < mesh.vertexCount; i++)
    {
        aircraftTempPos.push_back(Model.dataUntextured.positions[mesh.vertexStartIndex +
            i].x);
        aircraftTempPos.push_back(Model.dataUntextured.positions[mesh.vertexStartIndex +
            i].y);
        aircraftTempPos.push_back(Model.dataUntextured.positions[mesh.vertexStartIndex +
            i].z);
        aircraftColor.push_back(Model.materials[mesh.materialIndex].diffuseColor);
        aircraftColor.push_back(Model.materials[mesh.materialIndex].diffuseColor);
        aircraftColor.push_back(Model.materials[mesh.materialIndex].diffuseColor);
    }
    ColorizedMesh tempMesh = create_color_mesh(window, allocator, aircraftTempPos,
        aircraftColor);
    colorizedMesh.push_back(std::move(tempMesh));
}
```

2.2 Loading time and per-frame operations.

Loading: Load models, texture images, allocate descriptor sets, and create model mesh objects, update descriptor sets.

```
    SimpleModel Model = load_simple_wavefront_obj(cfg::kModelPath);
    // other code
    for(auto const& mesh:Model.meshes)
    {
        // other code
    }
```

per-frame: Command buffers are recorded for rendering the scene. Commands are submitted for execution using vkQueueSubmit(). Finally, the rendered images are presented to the screen using vkQueuePresentKHR().

2.3 Relevant screenshots

The final effect of task 1.2 is as shown in the Figure 1.

Figure 1: final effect of task 1.2

3 Anisotropic filtering

In ‘vulkan_window.cpp‘, add the following code in ‘create_device()‘ function to enable anisotropic filtering.

```
#if CURRENT_VERSION == ANISOTROPIC_FILTERING
    deviceFeatures.samplerAnisotropy = VK_TRUE;
#endif
```

In ‘vkutil.cpp‘, add the following code in ‘create_default_sampler()‘ function.

```
#if CURRENT_VERSION == ANISOTROPIC_FILTERING
    samplerInfo.anisotropyEnable = VK_TRUE;
    samplerInfo.maxAnisotropy = 16.0f;
#endif
```

The final result is as shown in Figure 2. When enabling anisotropic filtering and setting maxAnisotropy to 16.0f, textures in the distance appear relatively clearer.

(a) On (b) Off

Figure 2: The difference between enabling or disabling anisotropic filtering

4 Visualizing fragment properties

4.1 Utilization of texture mipmap levels

In the ‘mipmapVisualize.frag‘, I use ‘textureQueryLod(uTexColor, v2fTexCoord).x‘ to obtain the level of detail (LOD) value of the texture ‘uTexColor‘ at the texture coordinate ‘v2fTexCoord‘. Then, I extract the integer and fractional parts of this LOD value, where the integer part represents the mipmap level, and the fractional part is used for interpolation. Finally, I output the interpolated result.

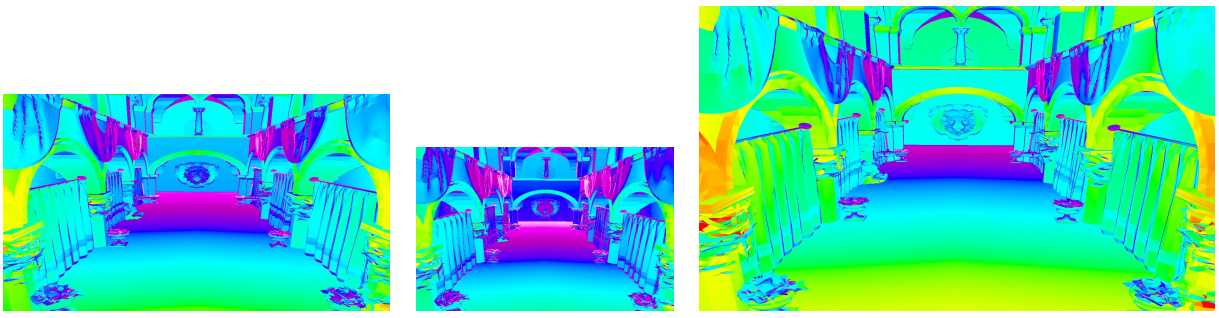
```
    float lod = textureQueryLod(uTexColor, v2fTexCoord).x;
    float mipmapLevel = floor(lod);
    float fraction = fract(lod);
    vec4 colorA;
    vec4 colorB;
```

```
    if(mipmapLevel == 0.0) {
        colorA = vec4(1.0, 0.0, 0.0, 1.0);
        colorB = vec4(1.0, 1.0, 0.0, 1.0);
    } else if(mipmapLevel == 1.0) {
        colorA = vec4(1.0, 1.0, 0.0, 1.0);
        colorB = vec4(0.0, 1.0, 0.0, 1.0);
    } else if(mipmapLevel == 2.0) {
        colorA = vec4(0.0, 1.0, 0.0, 1.0);
        colorB = vec4(0.0, 1.0, 1.0, 1.0);
    } else if(mipmapLevel == 3.0) {
        colorA = vec4(0.0, 1.0, 1.0, 1.0);
        colorB = vec4(0.0, 0.0, 1.0, 1.0);
    } else if(mipmapLevel == 4.0) {
        colorA = vec4(0.0, 0.0, 1.0, 1.0);
        colorB = vec4(0.5, 0.0, 0.5, 1.0);
    } else if(mipmapLevel == 5.0) {
        colorA = vec4(0.5, 0.0, 0.5, 1.0);
        colorB = vec4(1.0, 0.0, 1.0, 1.0);
    } else if(mipmapLevel == 6.0) {
        colorA = vec4(1.0, 0.0, 1.0, 1.0);
        colorB = vec4(1.0, 1.0, 1.0, 1.0);
    } else {
        colorA = vec4(1.0, 1.0, 1.0, 1.0);
        colorB = vec4(0.0, 0.0, 0.0, 1.0);
    }

    oColor = mix(colorA, colorB, fraction);
```

When the default mipmap mode is LINEAR, the rendering results are as shown in Figure 3 for the default size, magnification, and reduction. When enlarging the window, I noticed that lower Mipmap levels can be displayed, while when shrinking the window, I noticed that higher Mipmap levels can be displayed.

: , , , mipmap level



(a) dufault size (b) reduction (c) magnification

Figure 3: Mipmap mode is LINEAR

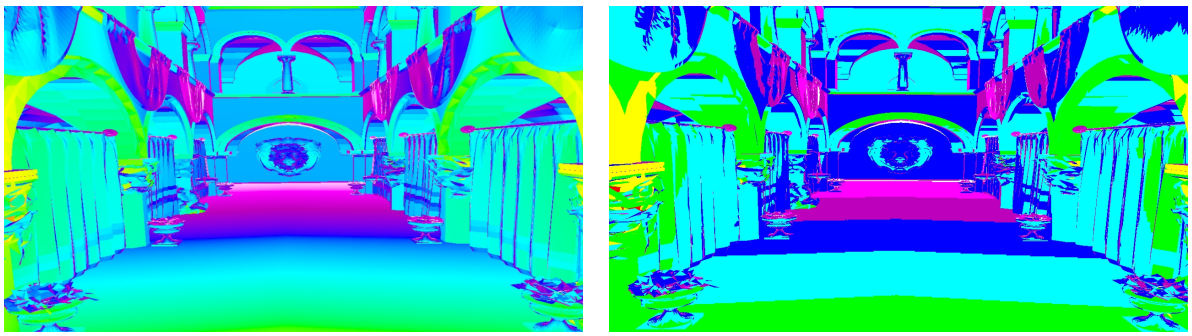
In the ‘create_default_sampler()‘ function in vkutil.cpp, different Mipmap modes can be obtained by modifying ‘samplerInfo.mipmapMode‘.

```
#if CURRENT_VERSION == MIPMAP_VISUALIZE_NEAREST
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_NEAREST;
#elif CURRENT_VERSION == MIPMAP_VISUALIZE_LINEAR
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
#else
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
#endif
```

The rendering results of the two Mipmap modes are shown in Figure 4. I noticed that the Mipmap transitions in LINEAR mode are smooth, while in NEAREST mode, the transitions are not smooth.

4.2 Fragment depth

In the ‘fragmentDepth.frag‘ shader, by calculating the result of ‘gl_FragCoord.z / gl_FragCoord.w‘, the normalized device coordinate depth in clip space is obtained, and then divided by 100 to obtain a relatively suitable



(a) LINEAR (b) NEAREST

Figure 4: Two different mipmap mode

range, which is output as the color of the fragment.

```
    // (X, Y, Z, W) -> (X/W, Y/W, Z/W, 1)
    float normalizedDepth = (gl_FragCoord.z / gl_FragCoord.w)/100.0;
    vec4 depthColor = vec4(normalizedDepth, normalizedDepth, normalizedDepth, 1.0);

    oColor = depthColor;
```

The resulting value is as shown in Figure 5.

Figure 5: Fragment depth

4.3 The partial derivatives of the per-fragment depth

In the ‘partialDerivativesFragmentDepth.frag‘ shader, the depth value is calculated using ‘gl_FragCoord.z / gl_FragCoord.w‘. Then, the partial derivatives are computed using the ‘dFdx()‘ and ‘dFdy()‘ methods. Finally, the derivatives are output as colors.

```
    float depth = gl_FragCoord.z / gl_FragCoord.w;

    float depthDerivativeX = dFdx(depth);
    float depthDerivativeY = dFdy(depth);
    vec3 depthGradient = vec3(depthDerivativeX, depthDerivativeY, 0.0);

    vec3 color = abs(depthGradient);
    color = clamp(color, 0.0, 1.0);
    oColor = vec4(color, 1.0);
```

The resulting value is as shown in Figure 6.

5 Visualizing overdraw and overshading

To visually clarify how drawn pixels are stacked together and the shading between different pixels, I enabled ColorBlend.

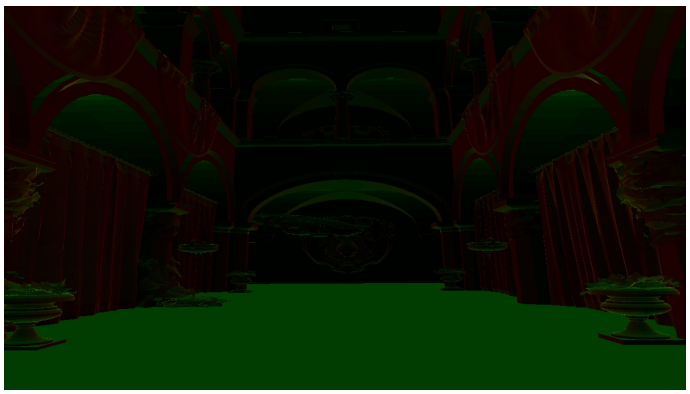


Figure 6: The partial derivatives of the per-fragment depth

```
#if CURRENT_VERSION == OVERDRAW_VISUALIZE || CURRENT_VERSION == OVERSHADING_VISUALIZE
    blendStates[0].blendEnable = VK_TRUE;
    blendStates[0].srcColorBlendFactor = VK_BLEND_FACTOR_ONE;
    blendStates[0].dstColorBlendFactor = VK_BLEND_FACTOR_DST_ALPHA;
    blendStates[0].colorBlendOp = VK_BLEND_OP_ADD;
    blendStates[0].srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
    blendStates[0].dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
    blendStates[0].alphaBlendOp = VK_BLEND_OP_ADD;
#else
    blendStates[0].blendEnable = VK_FALSE;
#endif
```

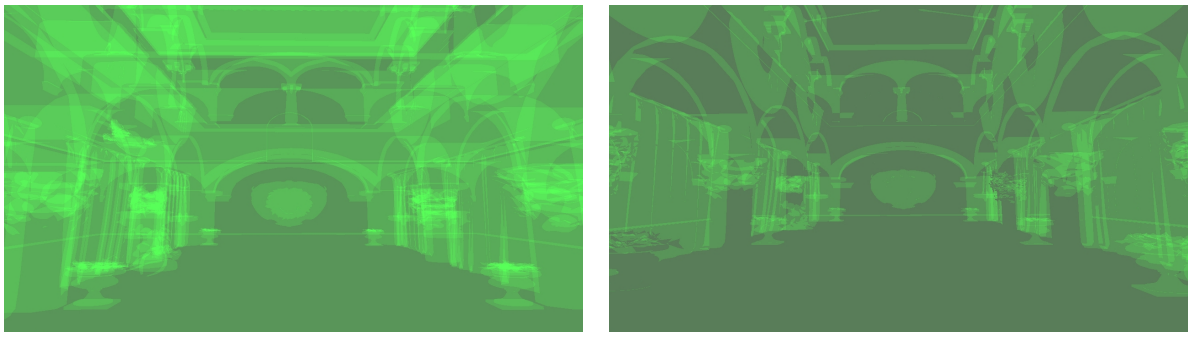
To prevent later-drawn pixels from covering previously drawn pixels in overdraw part, I disabled ‘depthTest‘ and ‘depthWrite‘ in the ‘create_pipeline()‘ function.

```
#if CURRENT_VERSION == OVERDRAW_VISUALIZE
    depthInfo.depthTestEnable = VK_FALSE;
    depthInfo.depthWriteEnable = VK_FALSE;
#else
    depthInfo.depthTestEnable = VK_TRUE;
    depthInfo.depthWriteEnable = VK_TRUE;
    depthInfo.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
    depthInfo.minDepthBounds = 0.f;
    depthInfo.maxDepthBounds = 1.f;
#endif
```

In the ‘overDraw_overShading.frag‘ shader, I set the color to dark green.

```
    oColor = vec4(0,0.1,0,1.0);
```

The resulting value is as shown in Figure 7. I can observe that, compared to Overshading, the result of Overdraw shows deeper colors in areas with overlapping surfaces, whereas Overshading only displays colors on surfaces that are retained after passing the depth test.



(a) Baseline Overdraw (b) Overshading

Figure 7: Baseline overdraw and Overshading

6 Visualizing Mesh Density

In the create_pipeline() method in main.cpp, add the geometry shader to it.

```
    lut::ShaderModule vert = lut::load_shader_module(aWindow, cfg::kVertShaderPath);
    lut::ShaderModule geom = lut::load_shader_module(aWindow,
        cfg::kMeshDensityShaderPath);
    lut::ShaderModule frag = lut::load_shader_module(aWindow, cfg::kFragShaderPath);

    VkPipelineShaderStageCreateInfo stages[3]{};
    stages[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    stages[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
    stages[0].module = vert.handle;
    stages[0].pName = "main";
    stages[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    stages[1].stage = VK_SHADER_STAGE_GEOMETRY_BIT;
    stages[1].module = geom.handle;
    stages[1].pName = "main";
    stages[2].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    stages[2].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    stages[2].module = frag.handle;
    stages[2].pName = "main";
```

And set pipeInfo.stageCount to 3 to specify the number of shader stages included in the pipeline.

```
    VkGraphicsPipelineCreateInfo pipeInfo{};
    pipeInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
    //pipeInfo.stageCount = 2;
    pipeInfo.stageCount = 3;
```

In the create_scene_descriptor_layout() function, modify bindings[0]. stageFlags to inform Vulkan that the pipeline includes both the vertex shader and the geometry shader stages.

```
//bindings[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
bindings[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_GEOMETRY_BIT;
```

Modify deviceFeatures.geometryShader in the create_device() function in vulkan_window.cpp to enable the device's geometry shader functionality.

```
    deviceFeatures.geometryShader = VK_TRUE;
```

Create a new geometry shader to calculate the density of the mesh using the average length of its three edges and pass this value to the fragment shader.

```
  vec3 v0 = (gl_in[0].gl_Position).xyz;
  vec3 v1 = (gl_in[1].gl_Position).xyz;
  vec3 v2 = (gl_in[2].gl_Position).xyz;

  float a = length(v1 - v0);
  float b = length(v2 - v1);
  float c = length(v0 - v2);

  float avgEdgeLength = (a + b + c) / 3.0;

  for(int i = 0; i < 3; ++i) {
    density = avgEdgeLength;
    gl_Position = uScene.projCam * gl_in[i].gl_Position;
    EmitVertex();
  }
  EndPrimitive();
```

In the fragment shader, set colors for high-density and low-density grids, and output the color of the current fragment based on the current density for both high and low-density regions.

```
    float normalized = 1.0 / (1.0 + exp(-1 * density));

    vec3 lowColor = vec3(0, 0, 0);
    vec3 highColor = vec3(0.8, 1.0, 0.8);
    vec3 finalColor = mix(lowColor, highColor, normalized);
```

```
6
7       oColor = vec4(finalColor, 1.0);
```

The final output resembles Figure 8. By observing the grid density, I can discern a smiling face pattern on the cloth hanging from the wall.

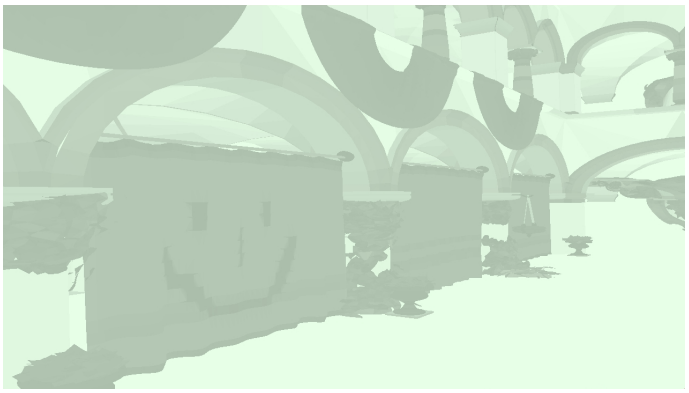


Figure 8: Mesh density

7 More overshading

……