Homework 2

For full code, see the end of each task section.

Task 1

Implementation

Given the task description and hints in lecture ("using linear interpolation"), the general idea of my implementation is to,

1. Get 8 color samples from 8 neighboring grid points in 3D LUT
2. Interpolate to the real value, given its neighbors
3. Return the interpolated sample as output.

Step 1

Corresponding code : `color_grading.frag`, line 22-69.

With given LUT examples in Pilot, it is obvious that, a LUT in Pilot should be a (16x16)x16 png, with its y-axis being Green value, and x-axis being a combination of Red and Blue values. Each color component is split into 15 grid cells, and has 16 grid values. A sRGB color with its 3 components in range [0, 1) can be mapped to LUT grid cells, and its 8 neighboring grid values can consequentially be found.

An exceptional case would be when a color component has the value of 1.0. In this case it would be mapped to the border of LUT, and has less than 8 neighboring grid points. To prevent the color from mapping out of LUT grid, certain clamping is needed.

During implementation I struggled a bit about branching. To deal with boundary conditions, some `if-else` statements are needed. As generally known, branching in GPU can significantly lower the performance, but I couldn't avoid them.

Another concern was about declaring 2 arrays (`rgbs[8]` and `colors[8]`) to store the grid values and their samples. I could have merged them into one array, but I was not sure how much performance optimization it would bring.

Step 2

Corresponding code : `color_grading.frag`, line 71-75.

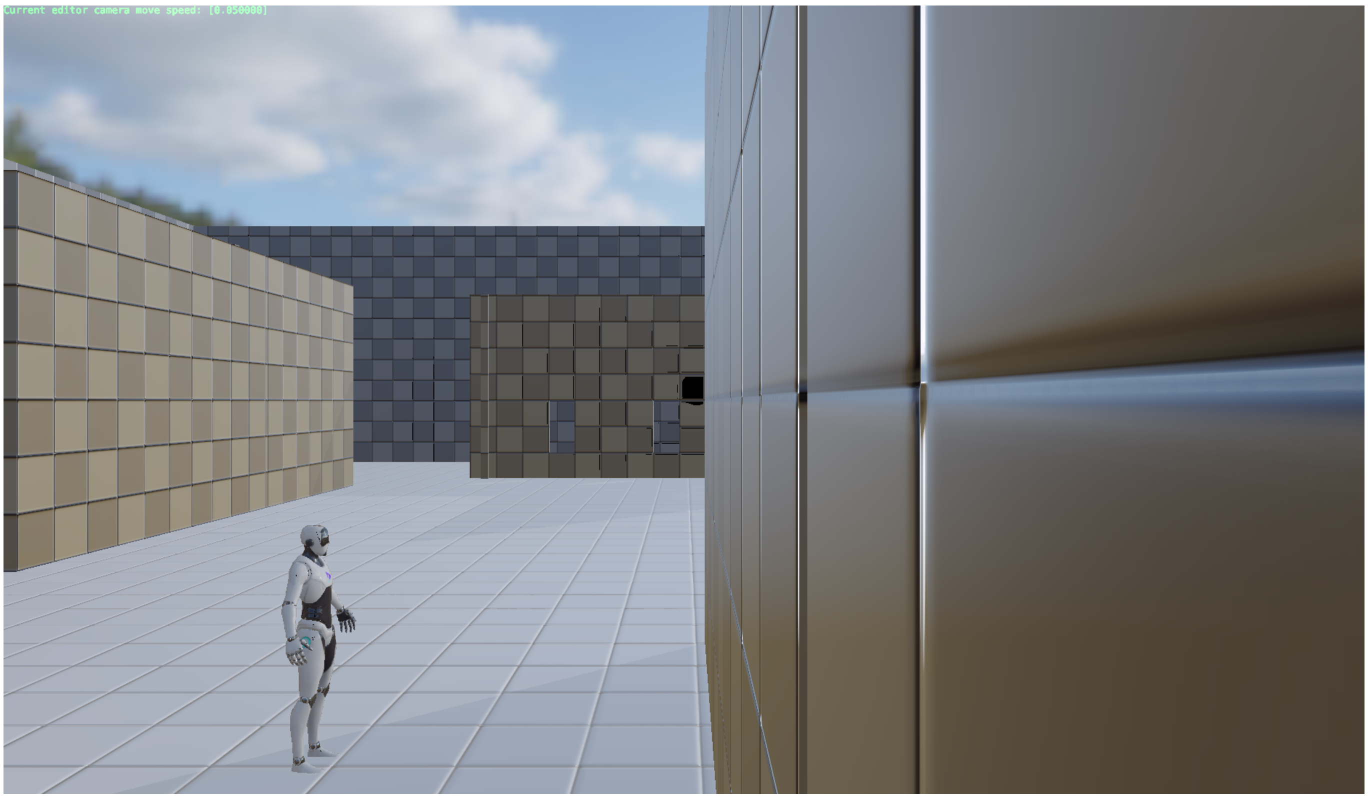
At first I misunderstood the interpolation. I thought it was simply a weighted average of all the 8 neighbors, and the weights were given by their distances to real value. The result had artifacts and when I looked back, a trilinear interpolation is different from weighted average. Obviously triliner interpolation works better for this task.

Step 3

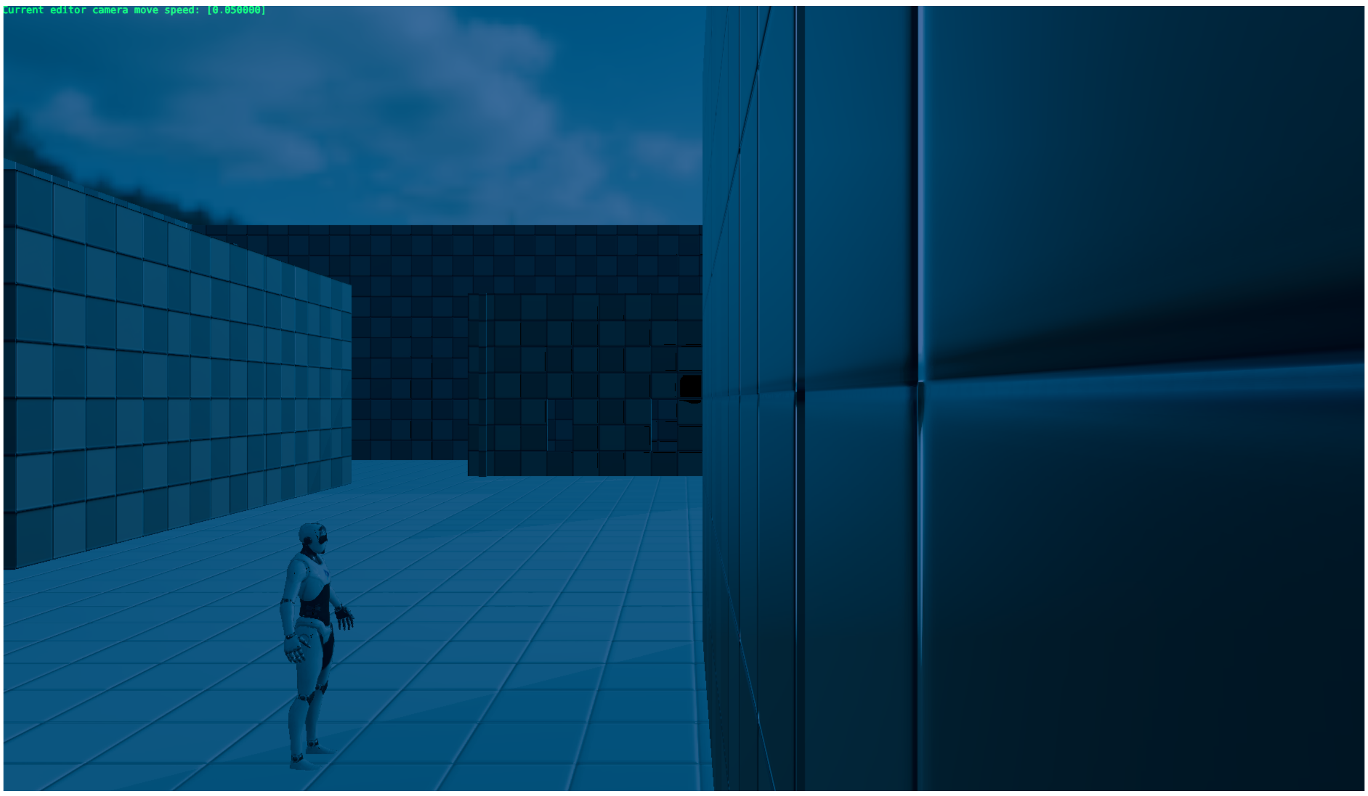
Corresponding code : `color_grading.frag`, line 76.

Task 1 - Result

Without color-grading:



With color-grading (color_grading_lut_05.png):



Task 1 - Code

Full Code of color_grading.frag

```
```

```
#version 310 es

#extension GL_GOOGLE_include_directive : enable

#include "constants.h"

layout(input_attachment_index = 0, set = 0, binding = 0) uniform highp
subpassInput in_color;

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

layout(location = 0) out highp vec4 out_color;

void main()
{
    highp ivec2 lut_tex_size =
textureSize(color_grading_lut_texture_sampler, 0);
    highp float _COLORS      = float(lut_tex_size.y);
    highp vec4 color       = subpassLoad(in_color).rgba;

    // Size of LUT in one dimension
    highp float size = 16.0;

    // sRGB values of 8 neighboring grid points
    highp vec3 rgbs[8];
    // And their corresponding color samples
    highp vec4 colors[8];

    // Get interpolation weights. Actual weight starting from base_color is
(1 - weight_next_x)
    highp float weight_next_r = fract(color.r * (size - 1.0));
    highp float weight_next_g = fract(color.g * (size - 1.0));
    highp float weight_next_b = fract(color.b * (size - 1.0));

    highp vec3 base_color = vec3(
        floor(color.r * (size - 1.0)) / size,
        floor(color.g * (size - 1.0)) / size,
        floor(color.b * (size - 1.0)) / size
    );
    highp vec3 next_color = vec3(0, 0, 0);
    if(color.r < 1.0) {
        next_color.r = base_color.r + (1.0 / size);
    } else {
        next_color.r = base_color.r;
    }
    if(color.g < 1.0) {
        next_color.g = base_color.g + (1.0 / size);
    } else {
        next_color.g = base_color.g;
    }
    if(color.b < 1.0) {
        next_color.b = base_color.b + (1.0 / size);
    } else {
        next_color.b = base_color.b;
    }

    // Sample 8 neighboring grid values from LUT
    rgbs[0] = vec3(base_color.r, base_color.g, base_color.b);
    rgbs[1] = vec3(next_color.r, base_color.g, base_color.b);
    rgbs[2] = vec3(base_color.r, next_color.g, base_color.b);
    rgbs[3] = vec3(base_color.r, base_color.g, next_color.b);
    rgbs[4] = vec3(next_color.r, next_color.g, base_color.b);
    rgbs[5] = vec3(next_color.r, base_color.g, next_color.b);
    rgbs[6] = vec3(base_color.r, next_color.g, next_color.b);
    rgbs[7] = vec3(next_color.r, next_color.g, next_color.b);

    for (int i = 0; i < 8; i++)
    {
        highp float v = rgbs[i].g;
        highp float u = rgbs[i].r / size + rgbs[i].b;
        colors[i] = texture(color_grading_lut_texture_sampler, vec2(u, v));
    }

    // Trilinear interpolation, note that mix in OpenGL: mix(x, y, a): x *
(1 - a) + y * a
    // First interpolate over r, then g, then b.
    highp vec4 color_res = mix(mix(colors[0], colors[1], weight_next_r),
mix(colors[2], colors[4], weight_next_r), weight_next_g);
    highp vec4 color_res2 = mix(mix(colors[3], colors[5], weight_next_r),
mix(colors[6], colors[7], weight_next_r), weight_next_g);
    highp vec4 result = mix(color_res, color_res2, weight_next_b);
    out_color = result;
}
```

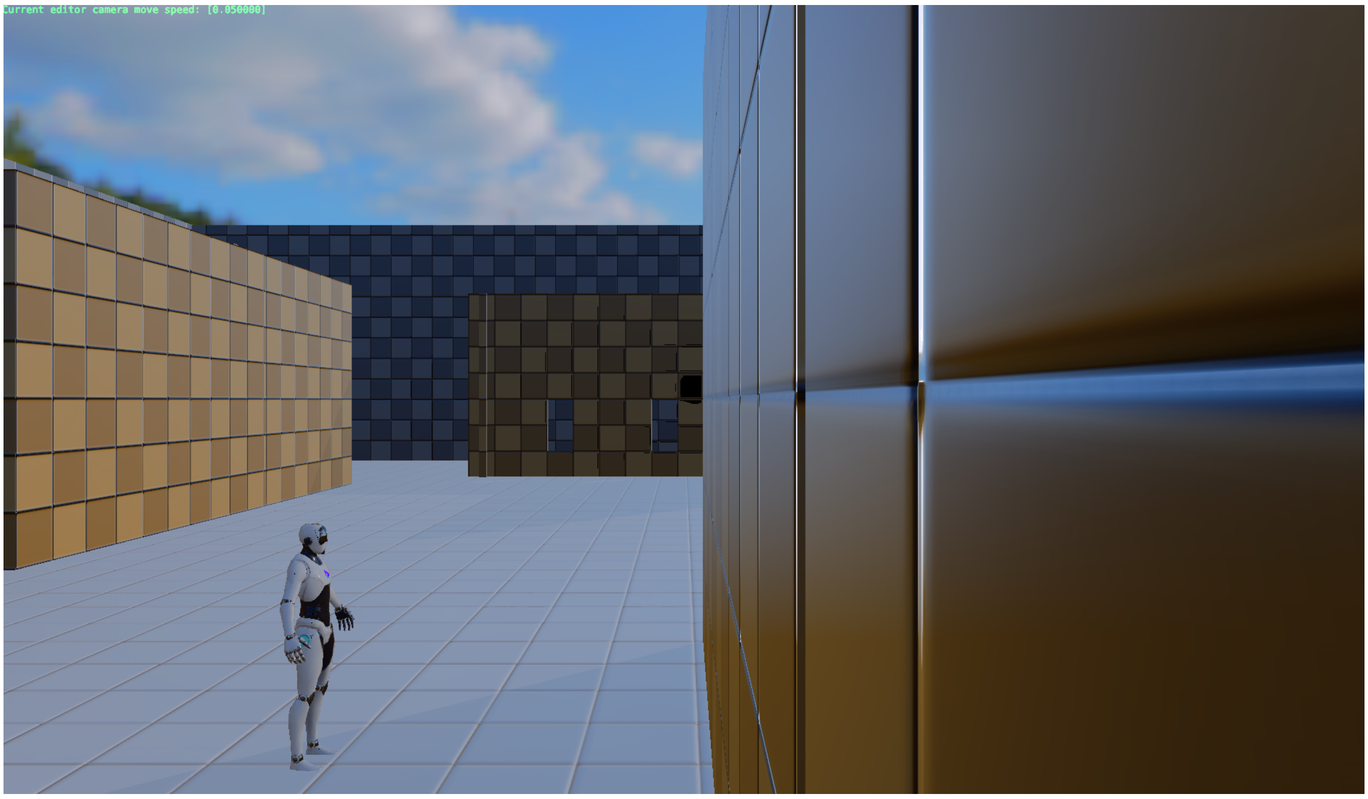
Task 2

Implementation

I found a free LUT (in .cube) online and adjusted it a bit to make the coloring stronger. It gives strong saturation in blue and orange, producing an "early-morning" vibe. Then I exported a 256x16 png with it.

Task 2 - Result

With customized color-grading (color_grading_lut_05.png):



Task 3

Implementation

I chose to implement Blooming effect based on Bloom - LearnOpenGL. The general process is listed below:

1. Get pixels with high brightness
2. Use a 1D Gaussian kernel to blur the highlighted pixels in x-direction
3. Use a 1D Gaussian kernel to blur the highlighted pixels in y-direction, then output to the next subpass

For the 3 steps I divided them into 3 subpasses. Usually the 2nd and 3rd subpasses could have been merged into one, but since I'm still new to Vulkan, I separated them.

Step 1

Corresponding code: `engine/source/runtime/function/render/source/vulkan_manager/passes/brightness_filter.cpp`

Fragment shader: `engine/shader/glsl/brightness_filter.frag`

Following `color_grading.cpp`, I created a new subpass in `brightness_filter.cpp`. This new subpass takes `attachments[_main_camera_pass_backup_buffer_odd]` as input, and outputs to another buffer, `_main_camera_pass_brightness_buffer`, that I created in advance.

Since tone mapping should take place after bloom effect, I put this subpass before tone mapping, and added dependencies in `main_camera.cpp`. Afterwards, I added this subpass into `PMainCameraPass::draw(...)` and `PVulkanManager::recreateSwapChain(...)` etc.

The fragment shader is only filtering the bright pixels with a high-pass filter. I also tried to "soften" the edge pf glowing parts a bit by linearly interpolating around the threshold.

Step 2

Correponding code: `engine/source/runtime/function/render/source/vulkan_manager/passes/gaussian_blur_x.cpp`

Vertex shader: `engine/shader/glsl/gaussian_blur_x.vert`

Fragment shader: `engine/shader/glsl/gaussian_blur_x.frag`

This subpass samples from the output of brightness_filter, `f`, and filters the sample with a 1D Gaussian kernel. For correct sampling, this subpass needs UV of each pixel in the scene.

The vertex shader generates UV of each pixel in full-screen coordinates, then the fragment shader translates the UV into viewport coordinates (without editor UI). This translation happens within the fragment shader. To pass the transformation into shader, a storage buffer is needed. Here I used the built-in `m_p_global_render_resource->_storage_buffer._global_upload_ringbuffer` so that it doesn't need to be initialized again.

The buffer is supposed to be updated when viewport dimensions change. Typically it happens during `updateAfterFramebufferRecreate(...)` of each render pass. During experiments, I found that the editor UI dimensions are not updated before UIPass draws, i.e. viewport coordinates cannot be caught in the very first rendering process. Also, the engine has a bug of not updating editor UI dimensions properly when maximizing/minimizing the UI window.

After trying to fix the forementioned bugs but failed (didn't want to change the pipeline too much), I decided to put the storage buffer updates in `PMainCameraPass::draw(...)`, i.e. updating the dimensions in every frame. This slows down the performance hugely and brings a lot of rendering artifacts (flickering due to I/O). Till now I haven't found a better and easier way to solve this issue.

After passing the storage buffer into fragment shader, I passed a sampler using the result of previous pass, `_main_camera_pass_brightness_buffer`, as the texture to be sampled. Then I implemented a 1D Gaussian kernel with the sampled RGB values, to blur the image on x-direction.

Step 3

Correponding code: `engine/source/runtime/function/render/source/vulkan_manager/passes/gaussian_blur_y.cpp`

Vertex shader: `engine/shader/glsl/gaussian_blur_x.vert` (yes, it's the same as previous)

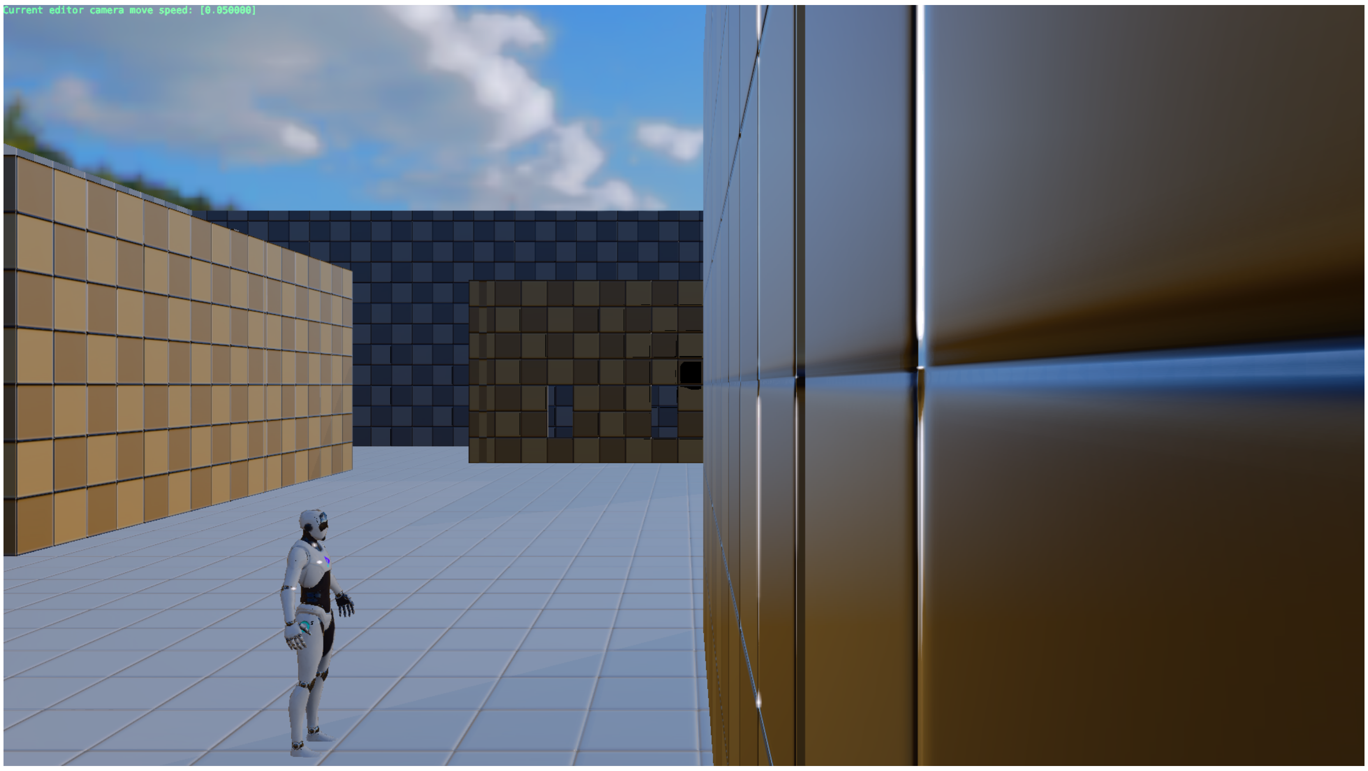
Fragment shader: `engine/shader/glsl/gaussian_blur_y.frag`

This subpass samples from the output of `gaussian_blur_x`, blur it in y-direction, then blend it with original scene, then output to another buffer . Compared to `gaussian_blur_x`, the biggest is color blending. Here I write the result into `attachments[_main_camera_pass_backup_buffer_odd]` with `VK_BLEND_OP_ADD`.

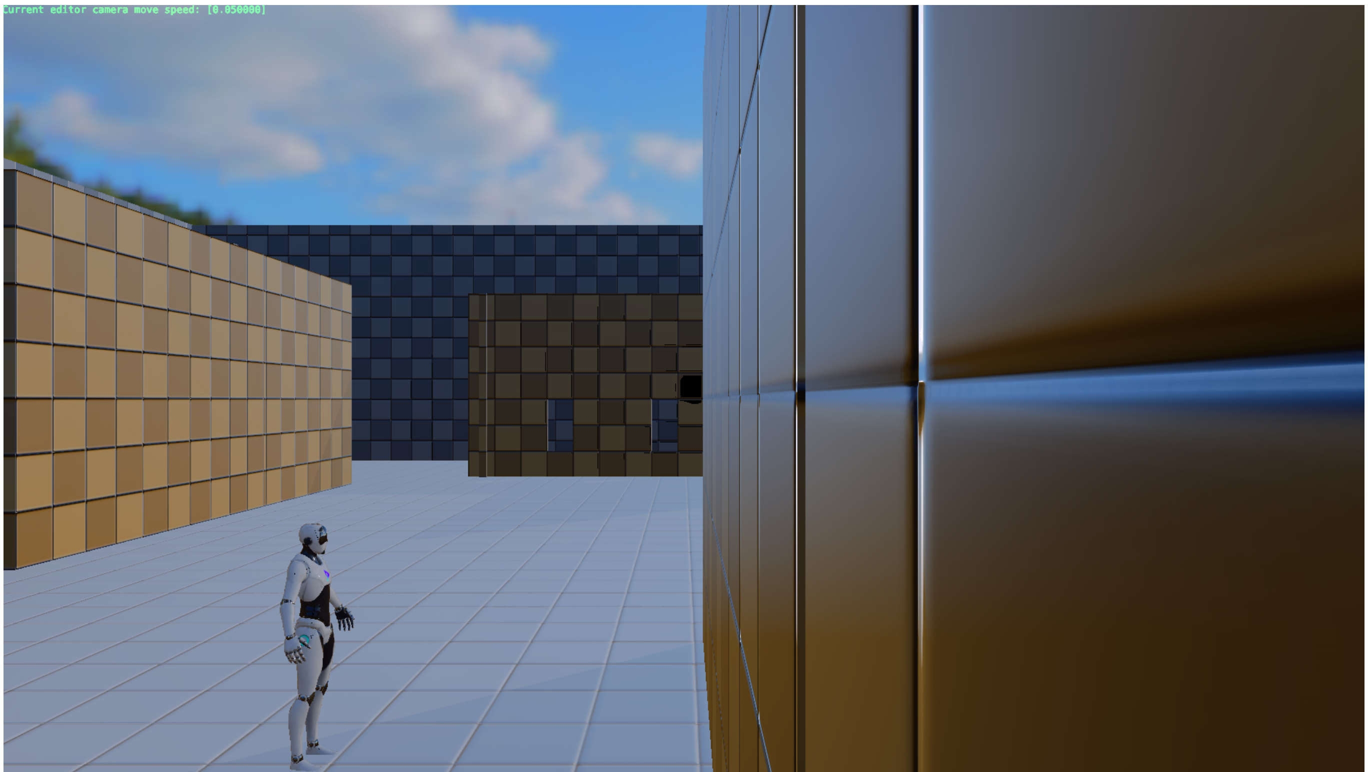
Task 3 - Result

The initial scene in the engine is outdoors with daylight, making it harder to observe the bloom effects. Still, it's obvious that the highlights (clouds, wall edges) in the scene are slightly glowing.

With blooming filter:



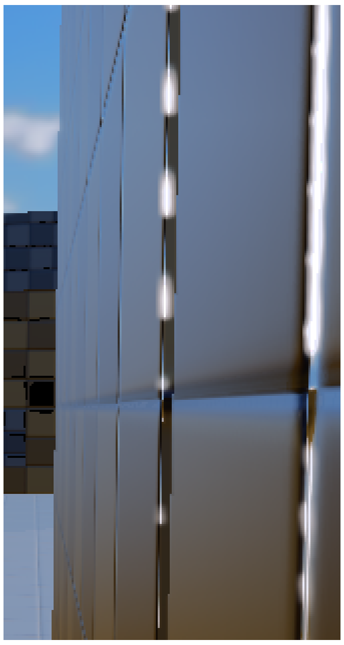
Without blooming filter:



Task 3 - Discussion

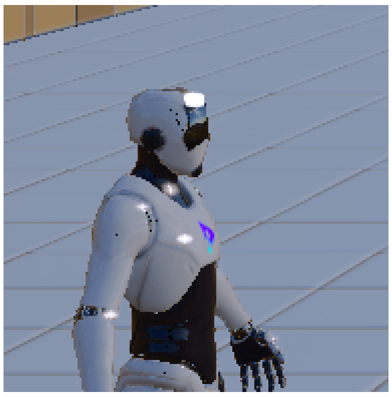
The general concept of bloom filter is simple, but the implementation using Pilot has been challenging. Here are a few thoughts that might make the performance smoother & better:

- An anti-aliasing pass will make the glowing parts more continuous at edges, e.g.,

  since the local brightness will have a smoother distribution. This can prevent the current glows from being fragmented.

- There are some flickering artifacts, especially on the character's body, e.g. on the character's head:

  I tried to eliminate it but failed. I suspect this artifact comes from sampling, but not sure what's the exact cause.

- Initialization of engine can be optimized by adding an inital viewport width/height. Then all the render passes which needs this information during rendering won't have to update their UBO every frame.

Task 3 - Some Code

Code fragments of brightness_filter.frag

```
...
...
```

```
void main()
{
    highp vec4 color       = subpassLoad(in_color).rgba;

    highp float brightness = dot(color.rgb, vec3(0.2126, 0.7152, 0.0722));
    if(brightness > 0.4) {  // here this limit should be smaller than
`minVal` in soften(...)
        out_color = vec4(color.rgb * soften(brightness), 1.0);
    } else {
        out_color = vec4(0.0, 0.0, 0.0, 1.0);
    }
}

// A "softer" blending of glow & darkness but failed
highp float soften(highp float br) {
    highp float minVal = 0.8;
    highp float maxVal = 0.95;
    if (br >= maxVal)
        return br;
    if (br <= minVal)
        return 0.0;
    return (br-minVal)*(maxVal/(maxVal - minVal));
}
```

Code fragments of gaussian_blur_x.frag

```
...
...
highp vec2 get_viewport_uv(highp vec2 full_screen_uv);

void main()
{
    highp float weight[5];
    weight[0] = 0.227027;
    weight[1] = 0.1945946;
    weight[2] = 0.1216216;
    weight[3] = 0.054054;
    weight[4] = 0.016216;

    highp vec2 sample_uv = get_viewport_uv(in_texcoord.xy);

    highp float intensity = 0.5;
    highp float range = 1.0;

    highp float tox = 1.0 / float(textureSize(scene_sampler, 0).x) * range;
    highp float toy = 1.0 / float(textureSize(scene_sampler, 0).y) * range;

    highp vec2 tex_offset = vec2(tox, toy);
    highp vec4 sampled_color = texture(scene_sampler, sample_uv).rgba;

    highp vec3 result = texture(scene_sampler, sample_uv).rgb * weight[0];
    for(int i = 1; i < 5; ++i)
    {
        result += texture(scene_sampler, sample_uv + vec2(tex_offset.x *
float(i), 0.0)).rgb * weight[i] * intensity;
        result += texture(scene_sampler, sample_uv - vec2(tex_offset.x *
float(i), 0.0)).rgb * weight[i] * intensity;
    }

    out_color = vec4(result, 1.0);

}

highp vec2 get_viewport_uv(highp vec2 full_screen_uv)
{
    highp vec2 editor_ratio = editor_screen_resolution.zw /
screen_resolution.xy;
    highp vec2 offset = editor_screen_resolution.xy / screen_resolution.xy;
    highp vec2 viewport_uv = full_screen_uv.xy * editor_ratio + offset;

    return viewport_uv;
}
```