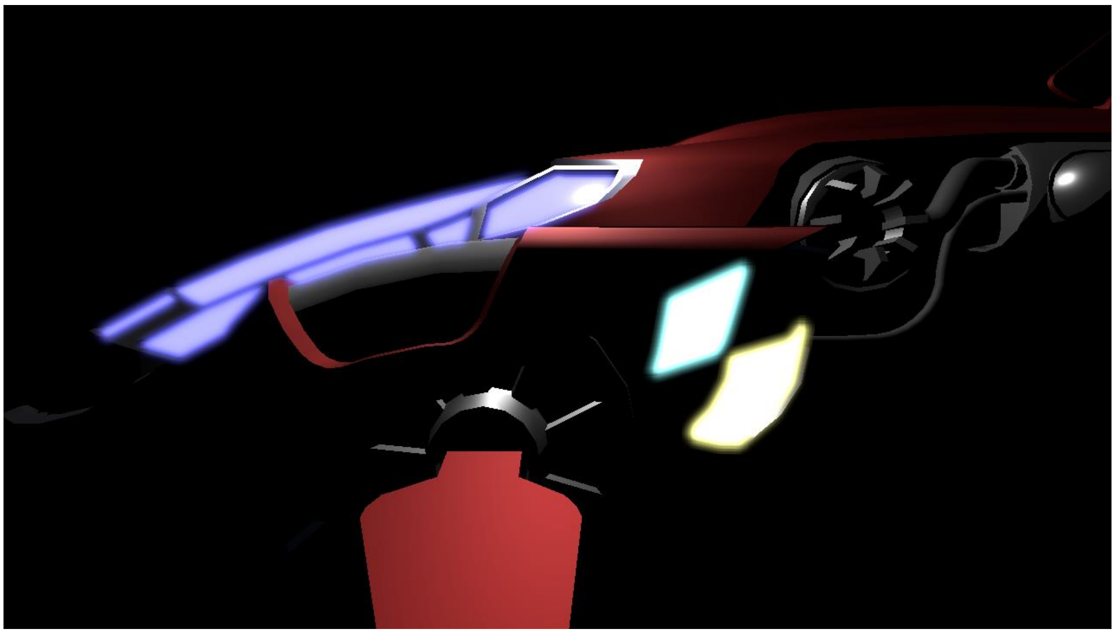
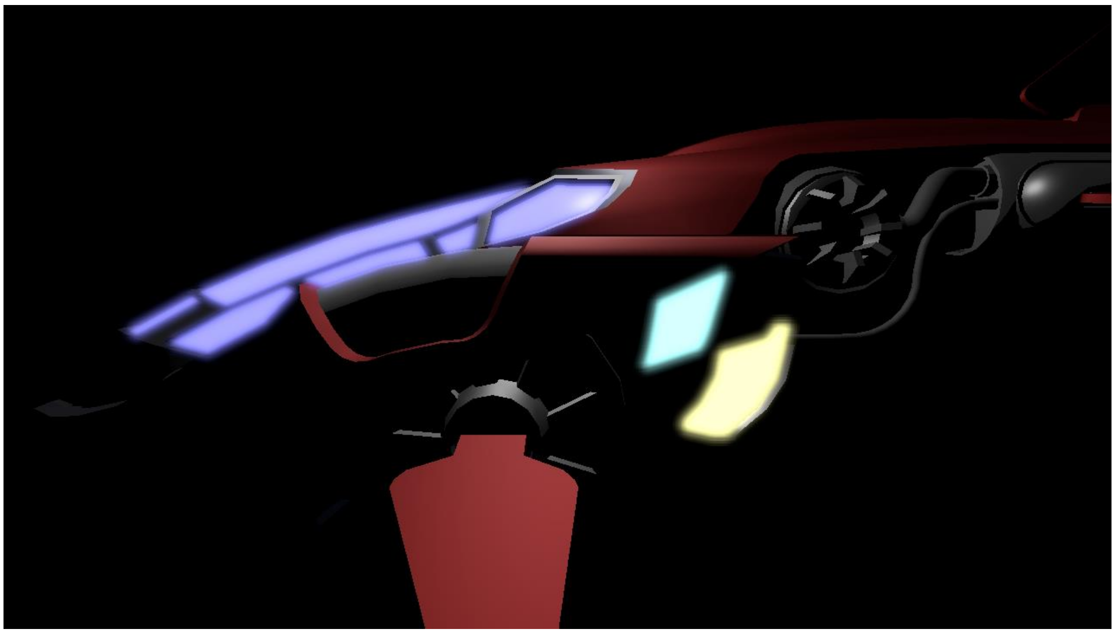
1.1Render-To-Texture Setup

synchronization: To achieve synchronization between swapchains, I've used two semaphores and one fence. The semaphores are used to signal when a swapchain image is ready to be presented and when it has finished being rendered, while the fence is used to wait for completion of a specific command before continuing with the next frame.
For synchronization between render passes, I've used subpass dependencies. Specifically, I've used a subpass dependency between the second and third render passes to ensure that the horizontal bloom pass finishes before the next pass begins.
Resources: For this implementation, I created two render passes. The first one is for outputting multiple gbuffers, including the albedo, normal, position, brightness, and depth gbuffers.
The final render pass combines the horizontally blurred image with the albedo color and the brightness color (which have been processed by the two bloom passes) to produce the final output.
The semaphores are needed to signal when a swapchain image is ready to be presented and when it has finished being rendered. The fence is needed to wait for completion of a specific command before continuing with the next frame. The subpass dependency and barrier are needed to ensure that the rendering process for each render pass is synchronized and that resources are properly processed before being used by subsequent passes.
Intermediate texture formats: For my implementation, I've used RGBA16F format for the gbuffer and RGBA16F format for the bloom passes. For the final output, I've used swapchain format . I chose these formats based on various factors such as memory bandwidth, performance, and image quality. Using RGBA16F format for the gbuffer and bloom passes saves memory bandwidth and improves performance,to achieve tone mapping while still maintaining sufficient precision for the required calculations. Using RGBA8UNORM format for the final output provides a good balance between image quality and performance.

1.2 Tone Mapping



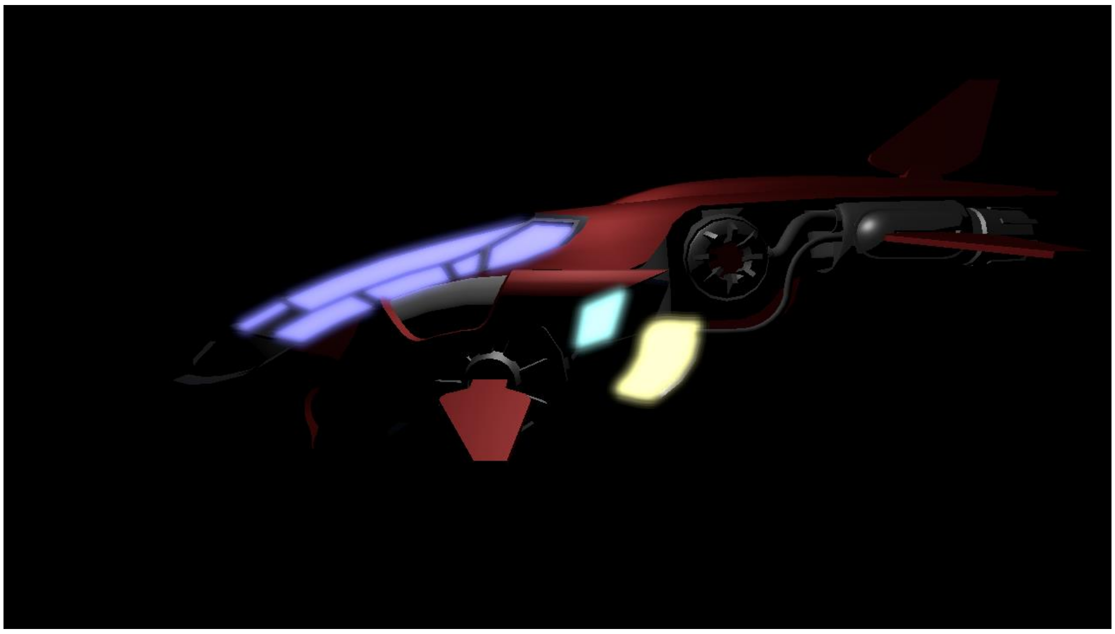
Before



After

As we can see, the emissive color now clearly have a better effect.

1.3 Bloom



results

outline:

I. **Create offscreen framebuffer and render pass**

Gbuffer render pass has four attachment:color(**VK_FORMAT_R16G16B16A16_SFLOAT**), normal(**VK_FORMAT_R16G16B16A16_SFLOAT**), position(**VK_FORMAT_R16G16B16A16_SFLOAT**) , brightness(**VK_FORMAT_R16G16B16A16_SFLOAT**) ,depth(**VK_FORMAT_D32_SFLOAT**), position and brightness are useless right now but I want to do deffered light so I reserve that.

Two bloom process has one bloom pass because they have the same attachment color(**VK_FORMAT_R16G16B16A16_SFLOAT**).

Offscreen render pass has two attachment:color(VK_FORMAT_R16G16B16A16_SFLOAT) and depth(VK_FORMAT_D32_SFLOAT).

II. **Pipeline prepare**

two different framebuffers and two different pipelines because I use **layout (constant_id = 0) const int blurdirection = 1;** to realize different bloom effect in just one same fragment shader.

III. **uniform buffer and descriptor set prepare**

The first pass has the same descriptor set and things as single render pass: SceneUniform, LightUniform, ColorUniform,bloom pass has one descriptor set: BloomUniform.

IV. **Command buffer**

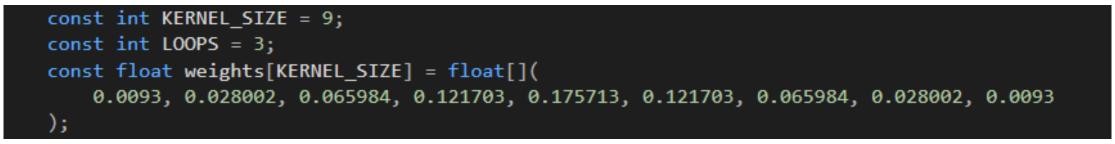
The clear value will depend on how many color attachments one pass will have,other steps are same as previous task.

The weights are hardcoded directly into the shader code and are not dynamically set at runtime.

To make the weights available to the shader, they could be passed as a uniform buffer object (UBO) to the shader. This approach would allow for the weights to be easily modified by the application at runtime without the need to recompile the shader code. In the provided code, the **ubo.blurStrength** uniform value is used to adjust the strength of the blur effect, but the weights themselves are hardcoded.

1.4 Performance

To measure the performance of the bloom implementation in the Vulkan project, I used a combination of Vulkan API profiling tools and Nvidia Nsight and vkCmdWriteTimestamp.I observed that increasing the number of passes/samples led to a significant increase in computational cost and a corresponding decrease in performance. The performance change will depend on the hardware capabilities of the machine, such as memory bandwidth and cache sizes. What's more, Graphics card parameters may affect screen flicker, especially when processing high dynamic range (HDR) images.I use smaller core sizes and cycle counts can improve performance, especially for lower-end graphics cards (RTX 2060).I enable precision highp float to have higher precision.My recommendation:



```
    const int KERNEL_SIZE = 9;
    const int LOOPS = 3;
    const float weights[KERNEL_SIZE] = float[](
        0.0093, 0.028002, 0.065984, 0.121703, 0.175713, 0.121703, 0.065984, 0.028002, 0.0093
    );
```

Blurscale:1.1f
Blurstrength:1.5f
My two graphic cards:
NVIDIA GeForce RTX 2060
NVIDIA GeForce RTX 3080
Changes in parameters after adjusting the filter: