**Post-processing Assignment report**

|  |  |
| --- | --- |
| Basic requirements | Checkmark with solid fill |
| Multiple effects at the same time in any order (full-screen and polygon) | Checkmark with solid fill |
| HSL gradient | Checkmark with solid fill |
| Two-pass Gaussian blur | Checkmark with solid fill |
| Feedback in blur effect | Checkmark with solid fill |
| Polygon post-processing | Checkmark with solid fill |
| Retro game mode | Checkmark with solid fill |
| Bloom | Checkmark with solid fill |
| Lens stars | Close with solid fill |
| imGUI | Checkmark with solid fill |
| 7 extra shaders | Checkmark with solid fill |
| Depth of field | Close with solid fill |

**1. The use of post-processing in a graphics application**

Post-processing applies an effect to a previously rendered image of the scene as explained by Steiner (2011). These effects can make the scene more realistic or add artistic touches. Post-processing effects work in two dimensions using the image data, and some additional information, like depth values and normals.

The image we see on the monitor comes from the front buffer (or viewport). When a frame is rendered, it is presented to the front buffer from the back buffer, which is the usual render target. However, the image can be rendered to a texture or a render target before the back buffer. This way additional image processing can be done in the pixel shader. The post-processing shader can access the entire image by offsetting the uv-s. Multiple passes can be done too, which requires more textures. This applies an effect to the whole screen, which is called full-screen post-processing.

It is possible to only apply the effects to smaller areas in the scene. One way to do this is by area post-processing, which is done around a known point in the world. The world point is converted into a screen point and the dimensions of the area are calculated. Finally, it is rendered to a quad, not the full screen.

Another way is polygon post-processing, where the effect is in the actual scene, like a model. A set of world points are transformed into 2D and their position is stored in the back buffer. Then it is rendered with the post-process.

**2. Specific techniques used**

**2.1 Gradient tint with changing hue**

First, two RGB colours are sent over to the GPU side. In the **Tint\_pp** shader they are converted into HSL colour space using functions. Then their hue is gradually increased by the frame time. It is also made sure that the hue doesn’t go out of the 0-360 range. Then it is converted back to RGB colour space and the gradient is calculated using the lerp function.

Finally, the scene texture is sampled and it is multiplicatively blended with the gradient colour.

**2.2 Box blur**

A simple one-pass blur effect can be implemented using the box blur algorithm. The surrounding neighbouring pixels and the pixel itself are averaged. It is unweighted, so all pixels contribute the same amount to the final image unlike in Gaussian blur.

A motion blur effect was also added to this shader (**Blur\_pp**). This technique blends the previous frames together using alpha blending. When motion blur is enabled, the alpha return value in the shader is set to 0.1. This is how much of the new frame is blended onto the existing one.

The motion blur doesn’t work the best if there are other post-processes on. It works well with most full-screen post-processes (except bloom), but it looks a bit strange with polygon and area post-processing. It is rendered before all the other post-processes. It can be switched on and off.

Multiple box blur effects can be added on top of each other with the user interface, and strength can also be varied.

**2.3 Underwater**

This shader (**Water\_pp**) uses both of the previous techniques and also wobbles the screen.

First, the UVs are manipulated using sin and the frame time constant that was sent over to the GPU. Then, the image is blurred using box blur, and multiplicatively blended with a bluish greenish RGB colour sent over.

**2.4 Multiple effects**

Up until now, the use of multiple textures was not needed since the individual post-processes were working when rendered straight to the back buffer. However, if we want to do more than one or multi-pass post-processes, then more textures are needed.

One way to do this, is using as many textures, as many passes are done, or as many post-processes are switched on at once. This works but it is not the most efficient or flexible.

The way this was done in this project is by swapping textures. A new texture was created and put into an array with the existing one. These two textures are switched as render targets after each post-process.

**2.5 Gaussian blur**

The image is blurred using the Gaussian curve. A kernel is calculated from the gaussian function values and it’s used as weights for the pixels. Example for kernels can be seen on Figure 1.

Chart, histogram

Description automatically generated

Figure 1 – Different kernel sizes.

If the Gaussian blur is done in two passes, then the kernel can be expressed as the outer product of two vectors. Doing the blur in two passes increases performance a lot.

First, a vertical pass is done, where the pixels are averaged in the positive and negative Y direction. The centre pixel is more heavily weighted than the surrounding pixels. This is rendered to a texture. Then using that texture as the shader resource, a horizontal pass is done, where the pixels in the X directions are averaged. And finally, it is rendered to the back buffer. These are implemented as two different post-processes (**GaussianVertical\_pp, GaussianHorizontal\_pp**).

Multiple gaussian blur effects can be added on top of each other with the user interface.

**2.6 Retro mode (Pixelated)**

This shader (**Pixelated\_pp**) divides the screen into rectangles. The size of the sides of the rectangles are decided by the passed value gPixelSize, and dy and dx. Then, these rectangles are filled with one colour that is sampled from their lower left corners. The relevant code snippet can be seen on Figure 2.

Text

Description automatically generated

Figure 2 - Pixelation shader

The pixel size can be changed with the user interface. The bigger the number, the smaller the pixels will be.

**2.7 Bloom**

The bloom effect makes the light areas look brighter and have a glow around them. For this effect, the current scene needs to be copied to a texture first. Then the areas above a specified brightness threshold are rendered to a texture. This texture is blurred using Gaussian blur. Finally, the blurred texture and the original scene are additively blended together.

The threshold can be modified with the user interface.

**2.8 Polygon post-processing**

The square wall opening can have various post-process effects. It can be changed with the user interface and also layered on top of each other. The wall with the complex openings has different post-processes for each opening.

New arrays were created for each opening, that contain the positions of the points surrounding the shapes. Then the post-processes are applied to them.

**2.9 Additional shaders**

### 2.9.1 Negative

This shader (**Negative\_pp**) inverts the colours of the scene. The scene texture is sampled and then that colour is subtracted from 1.

### 2.9.2 Posterization

This shader (**Posterization\_pp**) reduces the number of colours in the scene. It is done by limiting the colour precision, creating colour banding. The step size can be increased and decreased with the user interface. Based on Geeks3D shader library (2009).

### 2.9.3 Edge detection and Neon

The edge detection shader (**Edge\_pp**) outlines objects in the scene. A way to decide where an edge is without using the object geometry is looking at sudden colour changes. If it changes significantly in hue or brightness, then it could be an edge. The surrounding pixels are averaged, and the gradient magnitude is calculated, which decides if there is an edge or not.

The neon shader (**Neon\_pp**) uses the edge detection shader as its base, but also adds colour to it. The colours of the original scene are converted into HSV colour space and the lightness and saturation are increased to make the colours more vivid. Then it’s converted back to RGB and the outlined scene is multiplied with those colours. Based on Daniel Ilett’s article (2019).

### 2.9.4 Chromatic Aberration

Chromatic aberration happens when the lens fails to focus all colours to the same point. A similar effect can be achieved by offsetting the colour channels in the shader. The red and blue components of the scene texture are offset in different directions (**ChromaticAberration\_pp**). Based on Harry Alisavaki’s article (2017).

### 2.9.5 Paint effect

This paint effect (**Paint\_pp**) is implementing the Kuwahara filter. This is a noise reduction algorithm that preserves edges. It uses a kernel, like Gaussian blur, but it divides the window in four overlapping regions. The central pixel belongs to all regions. First, the colour variance of all the regions is calculated. The area with the lowest variance, so the most homogenous area is chosen, and the mean of those colours will be the colour of the central pixel (Ilett, 2019). Based on Shader Lab’s demo (2016).

### 2.9.6 Frost effect

This effect (**Frost\_pp**) makes it look like the user is looking through a frosted glass onto the scene. It uses a noise map that is sent over to the GPU and sampled using trilinear sampling. The noise texture was created in such a way that when you offset the post process pixel sampling, it creates a frosted glass effect. To make the glass seem blurrier, a simple box blur was also implemented. Based on Nvidia’s frost shader (2008).

The frequency can be changed with the user interface.

**3. Improvements and extensions**

An improvement on the already implemented shaders would be variable blur strength for the Gaussian blur, by increasing and decreasing the kernel. The kernel can be dynamically calculated on the C++ side and sent over to the shaders.

When increasing the strength for the box blur, it could be made smoother by averaging more surrounding pixels.

The gradient tint shader could be made more efficient by calculating the colour space conversions on the C++ side.

An obvious extension to the assignment would be being able to change the area post-process too. This would also require the user interface to be more sophisticated, since it is already a little cluttered with the various settings.

A good addition to the project would be implementing depth of field and screen space ambient occlusion. Implementing a realistic watercolour effect would be also really interesting.

**References**

Alisavaki, H. (2017) My take on shaders: Chromatic Aberration (Introduction to image effects part IV). Retrieved from: <https://halisavakis.com/my-take-on-shaders-chromatic-aberration-introduction-to-image-effects-part-iv/>.

Geeks3D (2009) [Shader Library] Posterization Post Processing Effect (GLSL). Retrieved from: <https://www.geeks3d.com/20091027/shader-library-posterization-post-processing-effect-glsl/>.

Ilett, D. (2019) Image Effects | Part 4 - Edgy Talk. Retrieved from: <https://danielilett.com/2019-05-11-tut1-4-smo-edge-detect/>.

Ilett, D. (2019) Image Effects | Part 6 - Painting Joy. Retrieved from: <https://danielilett.com/2019-05-18-tut1-6-smo-painting/>.

Nvidia (2008) NVIDIA Shader Library. Retrieved from: [http://developer.download.nvidia.com/shaderlibrary/webpages/shader\_library.html](http://developer.download.nvidia.com/shaderlibrary/webpages/shader_library.html#post_frost).

Shaders Laboratory (2016) Oil Painting. Retrieved from: <http://www.shaderslab.com/demo-63---oil-painting.html>.

Steiner, B. (2011) Post Processing Effects. Retrieved from: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.648.596&rep=rep1&type=pdf>.