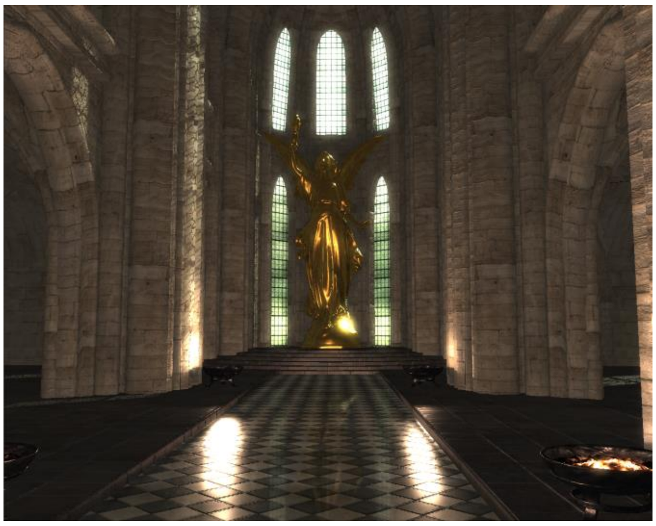
Tutorial 10: Tone Mapping and Bloom

This tutorial covers how to add Tone Mapping and Bloom post processing effects to an existing renderer.



This tutorial will continue on from the last by modifying the previously created code. It is assumed that you have created a copy of last tutorial's code that can be used for the remainder of this one.

Part 1: Tone Mapping

Standard texture and display formats have a fixed range of colours that they can support. As an example an 8bit textures colour values range from 0 to 255 where the brightest colour is (255,255,255) and is defined by the white point of the associated colour space. This is called a Low Dynamic Range (LDR) format. Real world environments can occupy a significant lighting range that LDR cannot efficiently handle. An environments lighting may be outside the bounds a LDR format can hold or it may occupy a sub section of the formats range resulting in reduced efficiency and loss of information.

High Dynamic Range (HDR) formats allow for an unconstrained range of colour luminance values. This is commonly done by using floating point numbers to represent linear colour values allowing a much greater range (i.e. where values can go outside the standard 0->1 range). Most output devices still cannot directly support HDR content. This requires converting between HDR and LDR. This process is called Tone Mapping. Tone Mapping relies on determining the range of HDR values and then determining an efficient conversion that then occupies the full LDR range.

Tone Mapping algorithms generally involve mapping based on average luminance and middle-grey. Middle-grey is the perceived distance between black and white as seen by the human eye. This is commonly 0.18 but varies based on a scenes light range. Middle-grey is a luminance value so Tone Mapping requires colour to be converted into luminance/chrominance values.

The relative luminance Y_r for a pixel (x, y) can be determined from the input luminance Y, the scene specific middle grey key d_g and the key scene luminance Y_a.

$$Y_r(x, y) = \frac{d_g Y(x, y)}{Y_a}$$

The scene specific middle-grey can be dynamically generated based on the key scene luminance Y_a.

$$d_g = 1.03 - \frac{2}{2 + \log_{10}(Y_a + 1)}$$

The final tone mapped luminance can be calculated by controlling the maximum luminance. This allows for any luminance above the threshold to “burn” out.

$$Y' = \frac{Y_r(x,y)\left(1 + \frac{Y_r(x,y)}{Y_w^2}\right)}{1 + Y_r(x,y)}$$

where Y_w is the smallest luminance that should be mapped to pure white.

The key scene luminance is a representation of the range of luminance values for the image. A useful approximation is to calculate the average luminance of the image. Luminance cannot be directly added however it can if it is converted to logarithmic values.

$$Y_a = \frac{1}{N} e^{\sum_{x,y} \log_e(0.00000001 + Y(x,y))}$$

where the 0.00000001 term is simply to avoid issues with input values of 0.

The final tone mapped colour can be calculated by dividing by the original luminance and then multiplying by the new value.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix}' = \begin{bmatrix} R \\ G \\ B \end{bmatrix} Y' \Big/ Y(x,y)$$

1. To implement tone mapping, we need 2 programs. The first is our existing post-processing program which we can use to calculate the final tone mapping luminance. However, we also need another program that will calculate the initial scene key value.

```
GLuint g_uiPostProcInitProgram;
```

2. First we need to create our new program that will initialise the luminance values and output them to a scene key value texture. This program will use the existing Vertex shader that we used previously for rendering a single full screen quad for the post-processing program. However, now we need a new Fragment shader that will calculate each pixel’s luminance and output it to a texture. This texture will be used to calculate the key scene luminance value.

```
#version 430 core

layout(binding = 7) uniform InvResolution {
    vec2 v2InvResolution;
};
layout(binding = 15) uniform sampler2D s2AccumulationTexture;

layout(location = 0) out float fYOut;

void main() {
    // *** Add luminance calculation code here ***
}
```

3. Within the Fragment shader the first thing we need to do is retrieve the current value for the pixel from the accumulation buffer. We can then calculate the luminance value by converting it to CIE XYZ colour space. We assume the input colour is in a linear sRGB colour space so we can use known conversion functions to calculate only the value of Y (as this represents luminance).

```
// Get UV coordinates
vec2 v2UV = gl_FragCoord.xy * v2InvResolution;

// Get colour data
vec3 v3AccumColour = texture(s2AccumulationTexture, v2UV).rgb;

// Calculate luminance
const vec3 v3LuminanceConvert = vec3(0.2126f, 0.7152f, 0.0722f);
float fY = dot(v3AccumColour, v3LuminanceConvert);
```

4. Now we need to convert the luminance into log luminance. This is because luminance cannot be correctly added and averaged. However, it can be approximated if converted to a logarithmic form. By outputting in log luminance we can then average each pixel's log luminance to get an average for the entire image. To avoid issues with calculating the logarithm of zero we ensure a valid positive offset is applied to all luminance values.

```
// Calculate log luminance
const float fEpsilon = 0.00000001f;
float fYoutTemp = log(fY + fEpsilon);
```

5. To calculate the average luminance, we can simply generate a set of mipmaps where the smallest mipmap will contain a single pixel with the average for the entire image. This can be used as the scene key luminance value. However, log luminance is a floating point value that has a large negative range. Many current GPUs don't correctly generate mipmaps for floating point texture formats so we instead have to use an integer one. This requires mapping the range of log luminance values to (0,1). This requires determining the possible range of log luminance values which is normally unbounded. Luckily we are using a tone map algorithm that defines a minimum white value that causes all values above it to burn out. We can use this white value as the upper bounds of the luminance range. The lower bounds are then simply the smallest possible luminance value which in this case is defined by the existing epsilon value. We can then map the existing log luminance to the range specified by the log of the maximum and minimum bounds. Once we have corrected the range we can then output it to texture.

```
const float fYwhite = 0.22f;
// Perform range mapping (optional: only needed with integer textures)
fYoutTemp = (fYoutTemp - log(fEpsilon)) / (log(fYwhite) - log(fEpsilon));

fYOut = fYoutTemp;
```

6. Now we need to update our existing post-processing program so that it uses the luminance output values. To do this we must first add the luminance texture as an input.

```
layout(binding = 16) uniform sampler2D s2LuminanceKeyTexture;
```

7. We now need to update the main body of the shader. Instead of getting a colour from the accumulation buffer and just directly outputting we now want to apply tone mapping to that colour. To do that we will add an additional function called "`toneMap`".

```
// Get colour data
vec3 v3RetColour = texture(s2AccumulationTexture, v2UV).rgb;

// Perform tone map
v3ColourOut = toneMap(v3RetColour, v2UV);
```

8. The first step that the tone mapping function must perform is to read in the value from the key luminance texture at the lowest resolution mipmap. Using GLSLs "**`texture`**" function we can pass an additional 3rd value that represents the bias that should be used when selecting a mipmap level. Here we will just supply a large value as the texture lookup just clips this to the maximum available level.

```
vec3 toneMap(vec3 v3RetColour, vec2 v2UV)
{
    // Get key luminance value
    float fYa = texture(s2LuminanceKeyTexture, v2UV, 1024).r;

    // *** Add remaining tone mapping operations here ***
}
```

9. Next we can perform the actual tone mapping calculations. This first requires undoing the range conversion that was previously applied to the luminance values so that they fit in an integer texture. Then we need to convert the log luminance value back to luminance by calculating its exponent. Using this value, we can then calculate the middle-grey value and then use it to determine the new luminance value for the current pixel based on a hardcoded max white luminance. The value read from the accumulation buffer is then modified by the new luminance to determine the new tone mapped colour. This new colour value can then be output from the shader.

```
// Perform range mapping (optional: only needed with integer textures)
const float fEpsilon = 0.00000001f;
fYa = (fYa * (log(fYwhite) - log(fEpsilon))) + log(fEpsilon);

// Get exponent of log values
fYa = exp(fYa);

// Calculate middle-grey
float fDg = 1.03f - (2.0f / (2.0f + log10(fYa + 1.0f)));

// Calculate current luminance
const vec3 v3LuminanceConvert = vec3(0.2126f, 0.7152f, 0.0722f);
float fY = dot(v3RetColour, v3LuminanceConvert);

// Calculate relative luminance
float fYr = (fDg / fYa) * fY;

// Calculate new luminance
const float fYwhite = 0.22f;
float fYNew = (fYr * (1.0f + (fYr / (fYwhite * fYwhite)))) / (1.0f + fYr);

// Perform tone mapping
return v3RetColour * (fYNew / fY);
```

10. Automatically calculating the middle-grey value requires performing a base 10 logarithm operation. This function is not found in GLSL however we can easily create our own function based on a simple logarithm rule.

```
float log10(in float fVal)
{
    // Log10(x) = log2(x) / log2(10)
    return log2(fVal) * 0.30102999566374217366165225171822f;
}
```

11. Now we need to modify the host code so that it loads the new shaders. The existing post-processing shader doesn't require any further work however the new shader must be loaded during program initialisation. This program just uses the existing Vertex shader that passes through a full screen quad as used by the current post-processing shader. The Fragment shader for this program is the newly added one.

12. Once the shader program has been set up we now need to add initialisation code to create a new frame buffer for the luminance shader pass. This just requires a new framebuffer and a new texture to contain the output luminance values.

```
// Post processing frame buffer
GLuint g_uiFBOPostProc;
GLuint g_uiLuminanceKey;
```

13. In the initialise section we now need to setup the new variables. This just requires generating a new frame buffer and texture. We will then create the luminance texture with enough space for a complete mipmap chain. We then specify the texture filtering values. Since we will only be sampling from the highest mipmap level we don't need any texture filtering. However, we need to specify "`GL_LINEAR_MIPMAP_NEAREST`" in order for mipmaps to be correctly created. We also create the texture using "`GL_R16`" format as we only need a single channel and "`glGenerateMipmap`" doesn't work on certain hardware

```
// Create post-process frame buffer
glGenFramebuffers(1, &g_uiFBOPostProc);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOPostProc);

// Generate attachment textures
glGenTextures(1, &g_uiLuminanceKey);

// Setup luminance attachment
glBindTexture(GL_TEXTURE_2D, g_uiLuminanceKey);
int iLevels = (int)ceilf(log2f((float)max(g_iWindowWidth, g_iWindowHeight)));
glTexStorage2D(GL_TEXTURE_2D, iLevels, GL_R16, g_iWindowWidth, g_iWindowHeight);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

// Attach frame buffer attachments
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
 g_uiLuminanceKey, 0);

// Bind luminance texture
glActiveTexture(GL_TEXTURE16);
glBindTexture(GL_TEXTURE_2D, g_uiLuminanceKey);
glActiveTexture(GL_TEXTURE0);
```

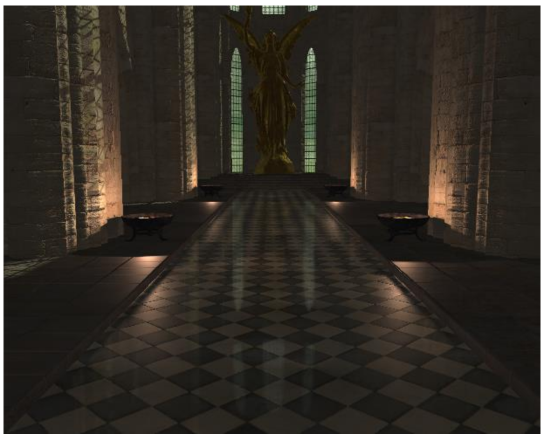
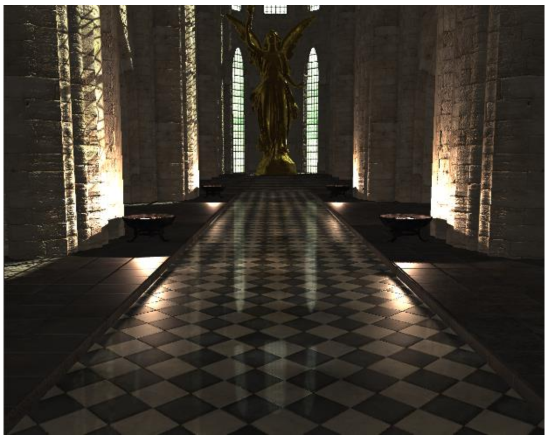
when using floating point formats. If on a platform where mipmap generation doesn't create NaN's within the higher levels of the mipmap then a more suitable format is "GL_R16F". When using a floating point format such as this then the range modifications in the post-processing shaders are not necessary. Once the texture has been set up we then bind it to the framebuffer as the default colour output. As there is only one luminance key texture we can bind it once during initialisation and then reset the active texture unit to prevent future texture functions affecting the current unit.

14. You should now add code to clean-up the newly added framebuffer, texture and shader program during program exit.

15. The final step in enabling tone mapping is to update the "GL_RenderPostProcess" function so that it first performs both Tone Mapping passes. This render pass now must do 2 render calls. First it must bind the luminance framebuffer, the luminance shader program and then perform the first pass. Once the first pass is complete we can then bind the default framebuffer and the second post-process shader program and perform the second and final pass. In order for this pass to work we need to bind the luminance key texture and then generate the complete mipmap chain.

```
void GL_RenderPostProcess()
{
    // Bind initialisation program
    glUseProgram(g_uiPostProcInitProgram);

    // Bind post-process frame buffer
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOPostProc);
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw full screen quad
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

    // Bind texture and generate mipmaps on luminance key texture
    glBindTexture(GL_TEXTURE_2D, g_uiLuminanceKey);
    glGenerateMipmap(GL_TEXTURE_2D);

    // Bind default frame buffer
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
    glClear(GL_COLOR_BUFFER_BIT);

    // Bind final program
    glUseProgram(g_uiPostProcProgram);

    // Draw full screen quad
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

    // Enable depth tests again
    glEnable(GL_DEPTH_TEST);
    glDepthMask(GL_TRUE);
 }
```

16. You should now be able to compile and run your program to see the effect of adding tone mapping.

Part 2: Bloom

Bloom effects are caused by real world lenses (either on cameras or eyes) being unable to focus perfectly. Incoming light is always convolved with the diffraction pattern caused by passing a point light through a circular aperture (Airy Disc). This causes bright areas to bleed into neighbouring regions.

Bloom effects involve checking each image value and only writing out those values that are greater than a predefined threshold. These output values are then convolved to match the effects of the Airy Disk function. Improved performance can be achieved by using a simpler Gaussian function to blur the output threshold values based on a downsized image. The blurred threshold values are then combined with the originals in the final output.

The threshold value used for Bloom effects should take into account the effects of Tone Mapping so it should be calculated on post Tone Map values. However ideally the contribution of Bloom should be added to the existing accumulation colour before the final Tone Map pass is applied. This requires calculating Tone Map values then getting Bloom values before performing the final Tone Map pass.

17. To perform Bloom, we must modify the newly created luminance Fragment shader so that it also outputs Bloom values as well as the existing log luminance.

```
layout(location = 1) out vec3 v3YBloomOut;
```

18. The Bloom pass needs to simply check each pixel and only output values for any pixel that’s brighter than some specified threshold. Checking a colours brightness simply requires checking its luminance value. The Bloom checks should be performed after Tone Mapping has already been applied so that the threshold maps to the new absolute output brightness range. This would normally require performing an additional shader pass after the first one but before the last that would calculate the Bloom effect. However we can improve on this by instead specifying the Bloom cut-off based on the Tone Map algorithm before it is even applied. As we are using a Tone Map algorithm that allows for setting minimum white luminance level we can actually use this value to specify our threshold. Since any value above this will be burnt to white this value signifies the upper bounds of luminance values

```
// Output bloom values
v3YBloomOut = (fY >= fYwhite * 2.95f)? v3AccumColour : vec3(0.0f);
```

that will occur after Tone Mapping. So by comparing luminance values against some linear function of the minimum white luminance we are actually performing the Bloom threshold based on approximated post Tone Map values. We can then add code to the end of the shader that simply checks the pixels luminance against the minimum white threshold. If it is above, then we output the current colour value otherwise we output zero.

19. Now we need to update the final post-processing shader so that it adds in the Bloom values. To do this we first need to add the Bloom texture to the next available texture binding.

```
layout(binding = 17) uniform sampler2D s2BloomTexture;
```

20. Within the shaders main body, we now need to combine the value from the Bloom texture to the existing colour value. This should be done after the accumulation value is read from texture but before the final Tone Mapping operation is applied to it. We read from the Bloom texture at a downsized resolution for improved performance. To support this we will specify a large bias which will cause texture reads to be from the highest mipmap level. We will later add host code that will control the actually maximum mipmap level that is read from.

```
vec3 bloom(vec3 v3RetColour, vec2 v2UV)
{
    // Perform bloom addition
    vec3 v3Bloom = texture(s2BloomTexture, v2UV, 1024).rgb;
    return v3RetColour + (v3Bloom * 0.98f);
}
```

21. Within the host code we now need to initialise the Bloom texture. This requires first adding a new texture to store the output of the Bloom pass. Ensure you add code to the initialisation section that correctly generates this new texture.

```
GLuint g_uiBloom;
```

22. During initialisation we now need to initialise the texture parameters. We will first allocate storage for the texture, as we don't need a full mipmap chain we only allocate enough levels based on the number that we need. As we don't need to generate a full mipmap chain, floating point texture formats generally work fine with the Bloom texture unlike with the luminance texture. We then set up texture filtering parameters similar to what we have used before. We need to use "`GL_LINEAR_MIPMAP_NEAREST`" so that we can at least create one mipmap level that we will use to downsize the Bloom texture. As we will then be reading from the downsized Bloom texture we need to specify magnification settings of "`GL_LINEAR`". We then use "`GL_TEXTURE_MAX_LEVEL` to specify the maximum mipmap level in the texture. This prevents "`glGenerateMipmap`" from generating mipmaps beyond this level. It also means that any GLSL texture lookups will be constrained to this maximum level.

```
// Setup bloom attachment
glBindTexture(GL_TEXTURE_2D, g_uiBloom);
iLevels = 2;
glTexStorage2D(GL_TEXTURE_2D, iLevels, GL_R11F_G11F_B10F, g_iWindowWidth,
g_iWindowHeight);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, iLevels - 1);
```

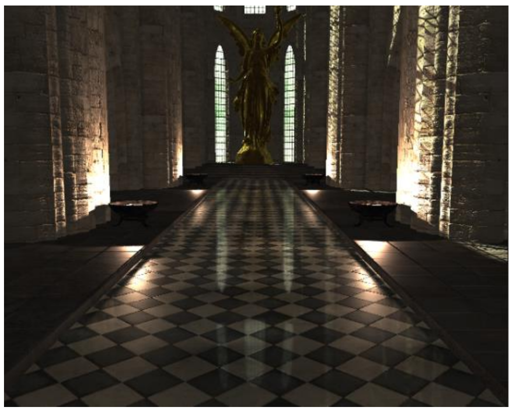
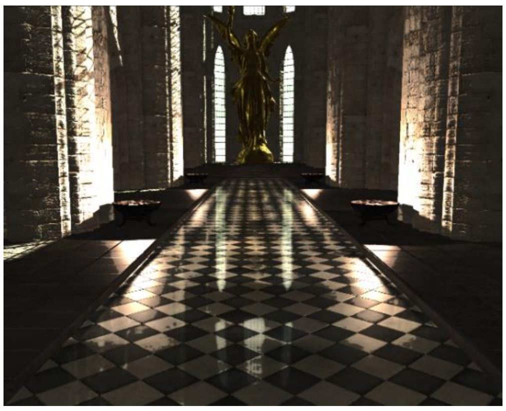
23. Now we need to bind the new texture to the existing framebuffer and then enable multiple framebuffer outputs. In this case we only have 2 colour outputs so we only need to enable the first 2 outputs.

```
// Attach frame buffer attachments
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
GL_TEXTURE_2D, g_uiBloom, 0);

// Enable frame buffer attachments
GLenum uiDrawBuffers[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, uiDrawBuffers);
```

24. Now we need to add code to "`GL_RenderPostProcess`" that downsizes the Bloom texture and then binds it to the correct texture location so that it is available for the final post-process pass. This code should be added immediately after the existing code binds the luminance texture and calculates its mipmaps.

```
// Bind texture and generate mipmaps on bloom texture
glActiveTexture(GL_TEXTURE17);
glBindTexture(GL_TEXTURE_2D, g_uiBloom);
glGenerateMipmap(GL_TEXTURE_2D);
```

25. You should now be able to compile and run your program to see the effect of adding a basic Bloom effect.

Part 3: Gaussian Blur

Many rendering and post-processing operations require using a spatial integral. This is often performed by applying a filter kernel over all pixels. If the filter kernel is independent of pixel location then the filter is considered “spatial-invariant”. An often used spatial-invariant filter function is the 2D Gaussian filter with variance σ^2 as denoted by:

$$\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The filter integrals are often approximated as a finite sum of n samples along the x and y directions of the 2D domain. This requires the evaluation of n^2 kernel samples which can become costly when executed for each pixel on the screen. The computation cost can be reduced by using spatial-invariant separable filters. With separable filters the two-variate filter kernel can be expressed as a product of two other filters each operating along only one of the filter coordinate axis. This breaks the operation into two passes with a total number of samples now only being $2n$.

The number of required samples can be reduced by using importance sampling instead of sampling the integration domain uniformly. The standard Gaussian filter has a ‘bell’ like shape where samples toward the centre are given higher weights than those near the edges. Since those samples near the outside of the filter region will have very low contribution to the final value they can be easily ignored. Importance sampling attempts to ensure samples are created in areas of high weight as opposed to areas of low contribution. With importance sampling the likelihood that a sample should be used is based on that samples weight. Therefore unlike with standard Gaussian filtering the contribution of each sample does not need to be scaled by its weight as this weighting has already been taken into account when calculating the probability that each sample will be used.

Using importance sampling for a Gaussian filter kernel with 5 samples creates the following sequence:

$$-1.282\sigma, -0.524\sigma, 0, 0.524\sigma, 1.282\sigma$$

This sequence can be used to calculate the offset for samples around a given pixels location in order to apply Gaussian filtering.

26. To improve our bloom effect we need to apply a Gaussian blur over the bloom texture. For this we will need a new shader to perform the Gaussian kernel operations. We will use the separable Gaussian filter which will require 2 separate passes, the first along the textures horizontal direction and the second over the vertical direction. Instead of creating 2 separate programs we will instead create just one but use 2 different subroutines for each of the passes.

```
GLuint g_uiGaussProgram;
```

27. Now we must create the Gaussian blur shader code. This code will run as a post-process and so it can use the same Vertex shader that previous post-processing shader programs have used. All we need is a new Fragment shader. This shader will use importance sampling to generate 5 different texture samples along the current separable pass direction. For the first pass this direction will be along the horizontal direction and the second pass this will be vertical. To keep things simple we will just create 2 subroutines that are used to calculate the sample offsets in one of the horizontal or vertical directions. These sample offsets will then be used to perform the texture sampling operations.

```
#version 430 core
layout(binding = 7) uniform InvResolution {
    vec2 v2InvResolution;
};
layout(binding = 17) uniform sampler2D s2InputTexture;

out vec3 v3ColourOut;

const float fGaussSigma = 2.5f;

subroutine vec2 SampleOffset(out vec2);
layout(location = 0) subroutine uniform SampleOffset SampleOffsetUniform;

layout(index = 0) subroutine(SampleOffset) vec2 horizSampleOffset(out vec2 v2D
u1)
{
    // Calculate horizontal sample offsets
    v2Du1 = vec2(0.524f * v2InvResolution.x * fGaussSigma, 0.0f);
    return vec2(1.282f * v2InvResolution.x * fGaussSigma, 0.0f);
}

layout(index = 1) subroutine(SampleOffset) vec2 vertSampleOffset(out vec2 v2Du
1)
{
    // Calculate vertical sample offsets
    v2Du1 = vec2(0.0f, 0.524f * v2InvResolution.y * fGaussSigma);
    return vec2(0.0f, 1.282f * v2InvResolution.y * fGaussSigma);
}

void main() {
    // Get UV coordinates
    vec2 v2UV = gl_FragCoord.xy * v2InvResolution;

    // Get sample offsets
    vec2 v2Du1;
    vec2 v2Du2 = SampleOffsetUniform(v2Du1);

    // Get filtered values
    vec3 v3Filtered = texture(s2InputTexture, v2UV - v2Du2).rgb +
                      texture(s2InputTexture, v2UV - v2Du1).rgb +
                      texture(s2InputTexture, v2UV).rgb +
                      texture(s2InputTexture, v2UV + v2Du1).rgb +
                      texture(s2InputTexture, v2UV + v2Du2).rgb;
    v3ColourOut = v3Filtered / 5.0f;
}
```

28. Now we need to modify the host code so that it loads the new shader. The existing post-processing shaders don't require any further work however the new shader must be loaded during program initialisation. This program just uses the existing Vertex shader that passes through a full screen quad as used by the current post-processing shaders.

29. Since the blur shader uses 2 passes we need to create 2 new framebuffers that we can use for each pass. The first pass will read in from the Bloom texture and then needs a new texture to store the intermediary results in. The second pass will use the second framebuffer to read in from the intermediary texture and then write the final values back into the Bloom texture.

```
GLuint g_uiFBOBlur;
GLuint g_uiFBOBloom;
GLuint g_uiBlur;
```

30. During initialisation we now need to create the framebuffers and intermediary texture. In order to improve performance of the blur pass we will do it at half resolution so once we bind the first framebuffer we will create the new texture using only half the window resolution and attach it as the output buffer. We will then bind the second framebuffer and set its output to the second mipmap level of the Bloom texture. As each mipmap level is half the resolution of the previous then this will correspond to half resolution processing. This is why we only created the Bloom texture with 2 levels.

```
// Create blur frame buffers
glGenFramebuffers(1, &g_uiFBOBlur);
glGenFramebuffers(1, &g_uiFBOBloom);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOBlur);

// Create blur texture
glGenTextures(1, &g_uiBlur);
glBindTexture(GL_TEXTURE_2D, g_uiBlur);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_R11F_G11F_B10F, g_iWindowWidth / 2,
 g_iWindowHeight / 2);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

// Attach frame buffer attachments
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, g_uiBlur, 0);

// Attach second frame buffer to bloom second mipmap level
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOBloom);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, g_uiBloom, 1);
```

31. Now we can modify the "`GL_RenderPostProcess`" function so that it uses the new blur pass to create the Bloom texture. Previously we just used a mipmap pass to calculate a lower resolution image that we used as the Bloom texture. Now we will manually create a lower resolution Gaussian blurred texture. Instead of the existing call to "`glGenerateMipmap`" for the Bloom texture we now replace it with code to clamp the texture so that only the first mipmap level can be accessed. This uses "`GL_TEXTURE_BASE_LEVEL`" to set the start mipmap level to the first level. It then uses "`GL_TEXTURE_MAX_LEVEL`" to also force the maximum mipmap level to the first level. This way all texture lookups must come from only the top mipmap level as that it is currently the only level with any data in it.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
```

32. Now we can perform the blur operation. To do this we will first set the viewport to half resolution and update the corresponding inverse resolution UBO to match. We then bind the first blur framebuffer and the Gaussian blur program. We can then set the subroutine uniform so that the horizontal shader pass is performed and then render the full screen quad. This will read the Bloom texture at its initial full resolution from the first mipmap level and then perform a horizontal blur on it and then write it out to the half resolution intermediary texture.

```
// Half viewport
glViewport(0, 0, g_iWindowWidth / 2, g_iWindowHeight / 2);
vec2 v2InverseRes = 1.0f / vec2((float)(g_iWindowWidth / 2),
(float)(g_iWindowHeight / 2));
glBindBuffer(GL_UNIFORM_BUFFER, g_uiInverseResUBO);
glBufferData(GL_UNIFORM_BUFFER, sizeof(vec2), &v2InverseRes,
GL_STATIC_DRAW);

// Bind blur frame buffer and program
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOBlur);
glUseProgram(g_uiGaussProgram);

// Perform horizontal blur
GLuint uiSubRoutines[2] = {0, 1};
glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &uiSubRoutines[0]);
glClear(GL_COLOR_BUFFER_BIT);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);
```

33. Now we need to perform the vertical blur pass. To do this we bind the second blur framebuffer and then set the shader subroutine to the vertical pass. We then bind the intermediary texture as the current texture input (the correct texture unit is still active from when we first bound the bloom texture). We can then render the full screen quad which will result in the final blur pass being written into the second mipmap level of the Bloom texture as this was what was bound to the framebuffer. Once we complete the render operation we then modify the Bloom texture so that now the minimum and maximum mipmap levels are set to the second mipmap level. This will cause all future texture reads from the Bloom texture to be read from the half resolution mipmap level which now contains the result of the blur operation.

```
// Perform vertical blur
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOBloom);
glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &uiSubRoutines[1]);
glBindTexture(GL_TEXTURE_2D, g_uiBlur);
glClear(GL_COLOR_BUFFER_BIT);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

// Bind second blur mipmap level as texture input
glBindTexture(GL_TEXTURE_2D, g_uiBloom);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 1);
```

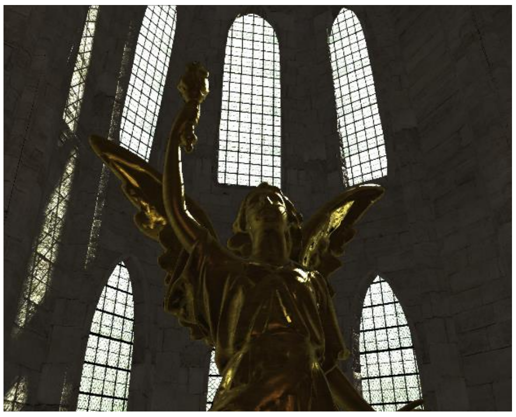
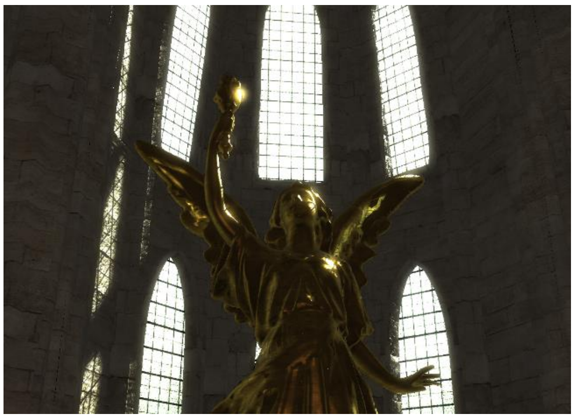
34. Now we have completed a single blur pass on the Bloom texture however to get a good looking Bloom we need to run multiple blur passes over the Bloom texture. Now that we have a half resolution Bloom texture we can just perform a loop where in each iteration we do the horizontal and vertical blur passes on the existing Bloom texture. Each iteration of the loop will further blur the Bloom which will cause the Bloom to appear to wrap further around objects when renderer to the screen.

```
const int iBlurPasses = 5;
for (int i = 1; i < iBlurPasses; i++) {
    // Perform horizontal blur
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOBlur);
    glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &uiSubRoutines[0]);
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

    // Perform vertical blur
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, g_uiFBOBloom);
    glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &uiSubRoutines[1]);
    glBindTexture(GL_TEXTURE_2D, g_uiBlur);
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);

    // Rebind bloom texture
    glBindTexture(GL_TEXTURE_2D, g_uiBloom);
}
```

35. Finally, once all blur passes have been completed and before the default framebuffer is bound we need to reset the viewport dimensions and update the inverse resolution UBO accordingly.

```
// Reset viewport
glViewport(0, 0, g_iWindowWidth, g_iWindowHeight);
v2InverseRes = 1.0f / vec2((float)g_iWindowWidth,
(float)g_iWindowHeight);
glBufferData(GL_UNIFORM_BUFFER, sizeof(vec2), &v2InverseRes,
GL_STATIC_DRAW);
```

36. You should now be able to run you program and observe the effect of the new Gaussian blurred Bloom effect.

Extra:

37. The Tone Mapping effect can be controlled by varying the value of the minimum white luminance value. Try changing the value of this and observe the effect on the output image (you should comment out the addition of Bloom in order to see the effect clearly). Add a user controllable variable that can be used to update the value of the minimum white luminance through a uniform variable.

38. Currently Bloom simply gets any value above 2.95 times the minimum white luminance setting. Try adjusting the Bloom threshold so that it is instead a different multiple of minimum white luminance (i.e. 0.5 -> 4.0 times white luminance). Observe the effect that this has on the Bloom effect. To make it more pronounced try multiplying the Bloom effect before adding it to the final colour. Try also experimenting with the value used to multiply the Bloom effect before adding to the final value.

39. The number of blur passes performed on the Bloom texture effects how much the Bloom effect covers nearby objects. Try adjusting the number of blur passes and observe the effect that it has on the final image. The more blur passes are performed the more computationally intensive the Bloom effect becomes. Try finding an optimum balance between performance and Bloom effect.

40. Currently the luminance texture uses an integer format to avoid issues with floating point textures on certain hardware. Try using a texture format of “GL_R16F” for the luminance texture and disable the range correction in both post processing shaders. If your hardware works correctly then this should still function correctly. However if there are hardware issues then you should see periodic black screens when looking in specific directions.