Computer Graphics II – Bloom

Kai Lawonn

Introduction

- Bright light sources and brightly lit regions difficult to convey as the intensity range of a monitor is limited
- One way to distinguish bright light sources is by making them glow, light bleeds around the light source
- Effectively gives the viewer the illusion these light sources or bright regions are intensely bright

Introduction

- This light bleeding or glow effect is achieved with a post-processing effect called bloom
- Bloom gives all brightly lit regions of a scene a glow-like effect
- An example* with glow:

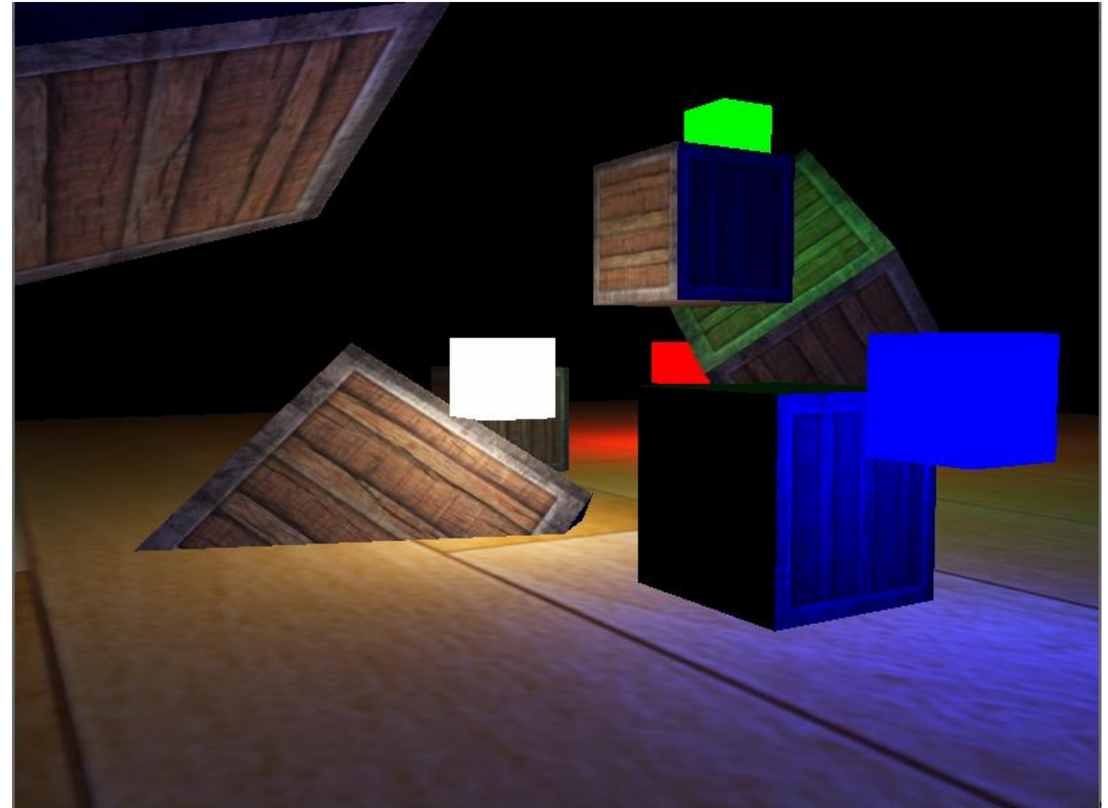
*https://commons.wikimedia.org/wiki/File:Elephants_Dream_-_Emo_and_Proog.jpg

Bloom

- Bloom works best in combination with HDR rendering
- Common misconception: HDR is the same as bloom
- Different techniques used for different purposes
- Possible to implement bloom with default 8-bit precision framebuffers just as it is possible to use HDR without the bloom effect
- It is simply that HDR makes bloom more effective to implement
- To implement Bloom, render a lighted scene and extract the scene’s HDR colorbuffer and an image of the scene with only its bright regions visible
- The extracted brightness image is then blurred and the result added on top of the original HDR scene image

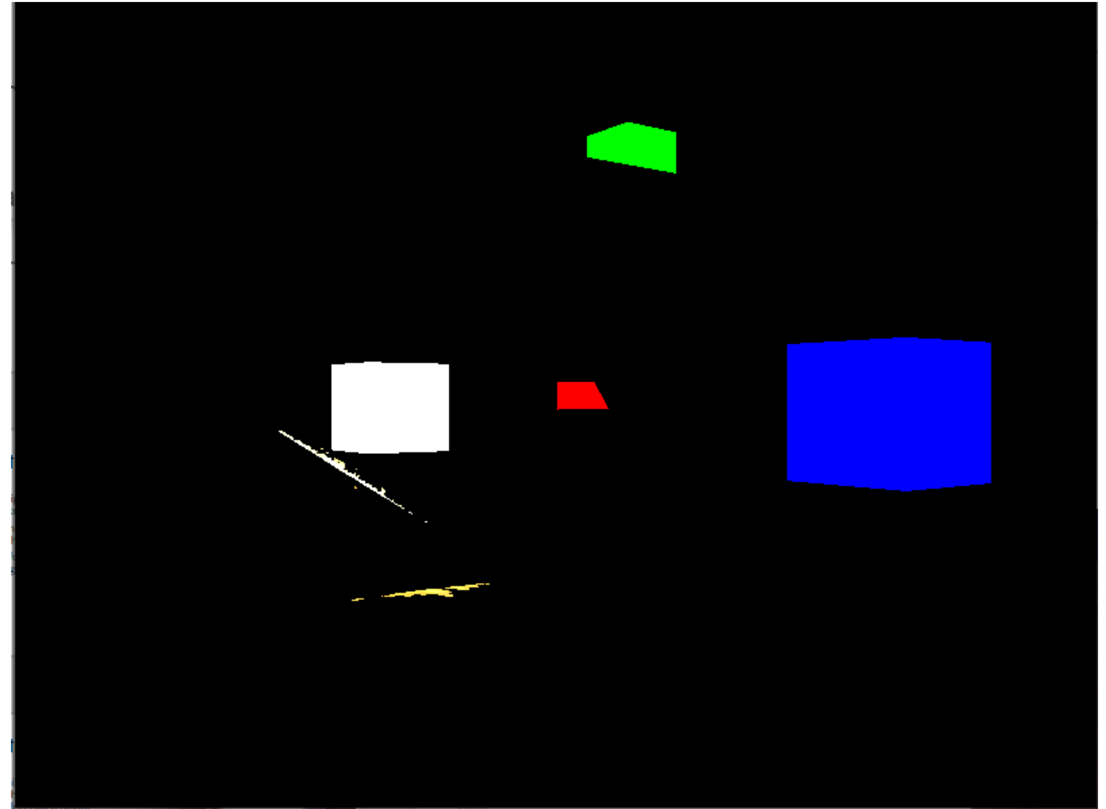
Bloom

- Illustrate this process step by step
- Render a scene with 4 bright light sources visualized as colored cubes
- Colored light cubes have a brightness values between 1.5 and 15.0
- If we were to render this to an HDR colorbuffer the scene looks as follows:



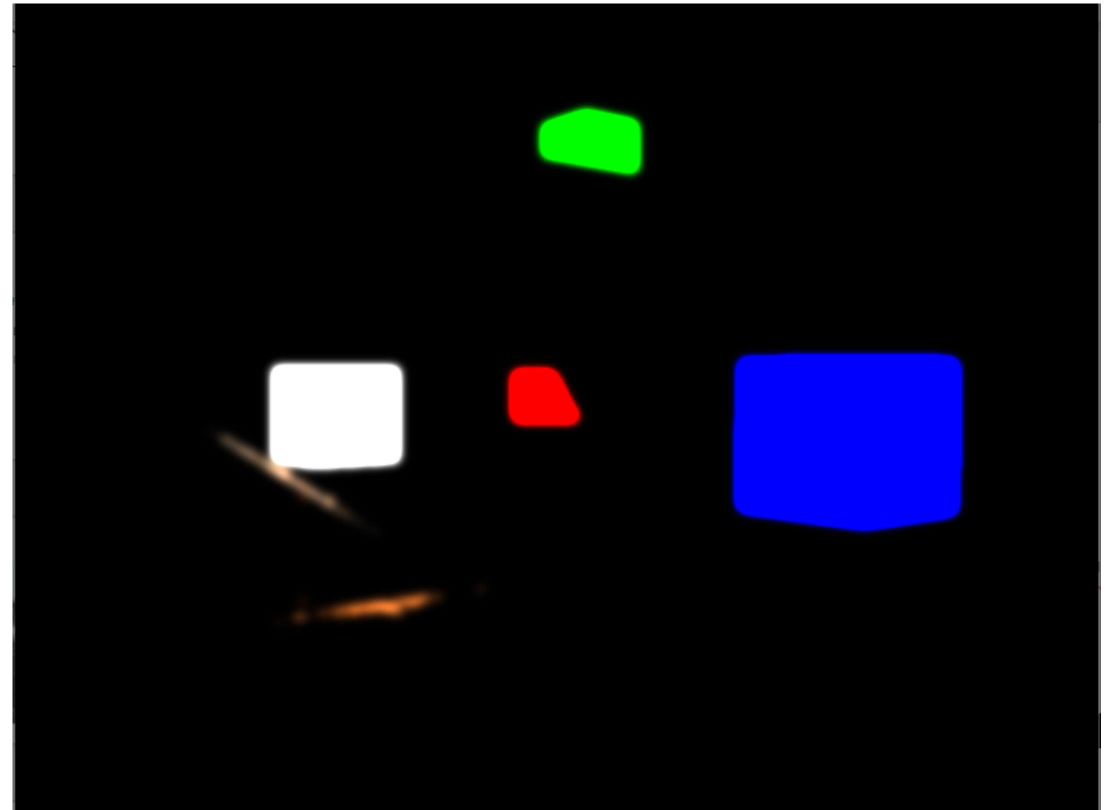
Bloom

- Take this HDR colorbuffer texture and extract all the fragments that exceed a certain brightness
- This gives an image that only shows the bright colored regions as their fragment intensities exceeded a certain threshold:



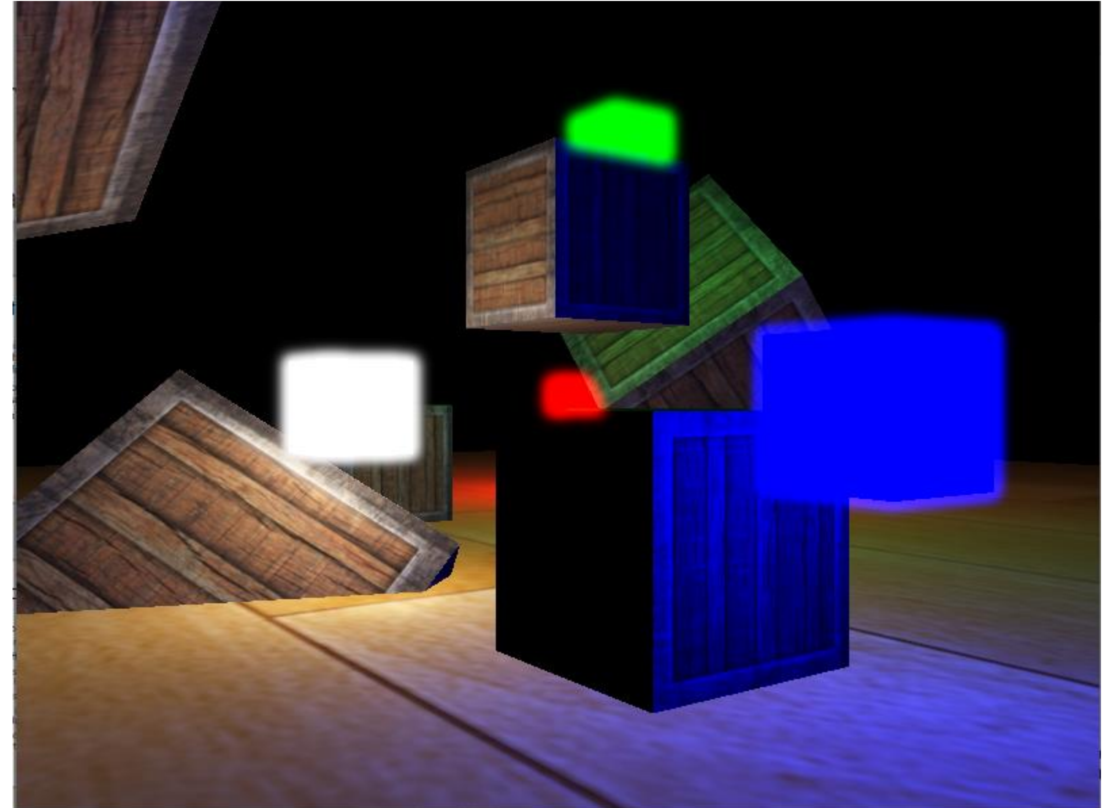
Bloom

- Take this thresholded brightness texture and blur the result
- The strength of the bloom effect is largely determined by the range and the strength of the blur filter used



Bloom

- Resulting blurred texture is what we use to get the glow or light-bleeding effect
- This blurred texture is added on top of the original HDR scene texture
- Bright regions are extended in both width and height due to the blur filter the bright regions of the scene appear to glow or bleed light

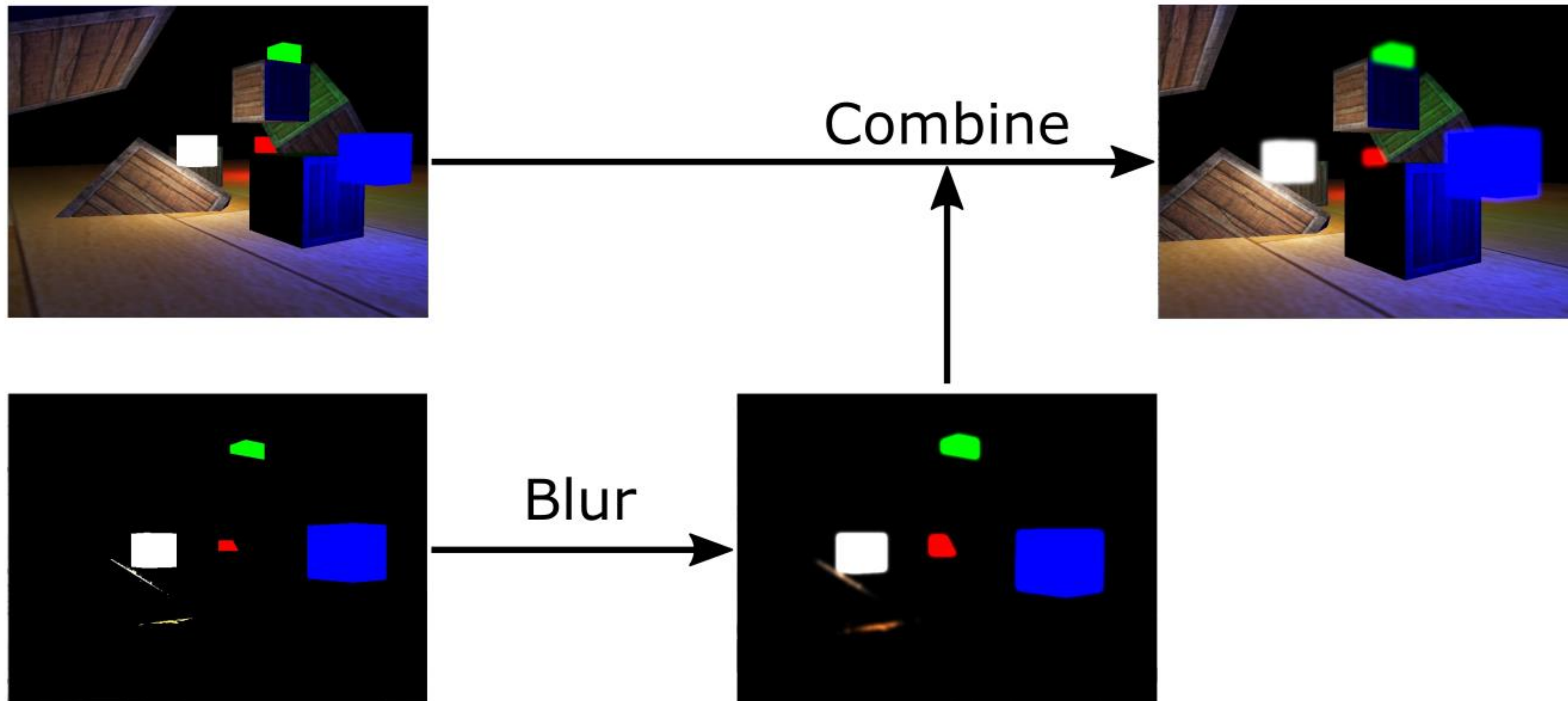


Bloom

- Bloom is not a complicated technique, but difficult to get exactly right
- Most of its visual quality is determined by the quality and type of blur filter used for blurring the brightness regions
- Simply tweaking the blur filter can drastically change the quality of the bloom effect

Bloom

- Following these steps give us the bloom post-processing effect
- The image summarizes the required steps for implementing bloom



Extracting Bright Color

Extracting Bright Color

- First extract two images from a rendered scene
- Could render the scene twice (rendering to different framebuffer with different shaders)
- Trick: Multiple Render Targets (MRT) allows to specify more than one fragment shader output (extract two images in a single render pass)

Extracting Bright Color

- Specifying layout location specifier before a fragment shader's output can control to which colorbuffer a fragment shader writes to:

```
layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;
```

Extracting Bright Color

- Works only if we have multiple places to write to
- Need multiple colorbuffers attached to the currently bound framebuffer object
- Framebuffers lecture: specify color attachment when linking a texture as a framebuffer's colorbuffer

Extracting Bright Color

- Use GL_COLOR_ATTACHMENT0 and GL_COLOR_ATTACHMENT1: two colorbuffers attached to a framebuffer object:

```
unsigned int hdrFBO;
glGenFramebuffers(1, &hdrFBO);
glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
unsigned int colorBuffers[2];
glGenTextures(2, colorBuffers);
for (unsigned int i = 0; i < 2; i++)
{
   glBindTexture(GL_TEXTURE_2D, colorBuffers[i]);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
   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 texture to framebuffer
   glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT0+i,GL_TEXTURE_2D,colorBuffers[i],0);
}
```

Extracting Bright Color

- Explicitly tell OpenGL to render multiple colorbuffers via glDrawBuffers (otherwise OpenGL only renders to the first color attachment ignoring all others)
- Passing an array of color attachment enums:

```
unsigned int attachments[2] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2, attachments);
```

Extracting Bright Color

- Fragment shader uses the layout location specifier the respective colorbuffer is used to render the fragments to
- Saves extra render pass for extracting bright regions:

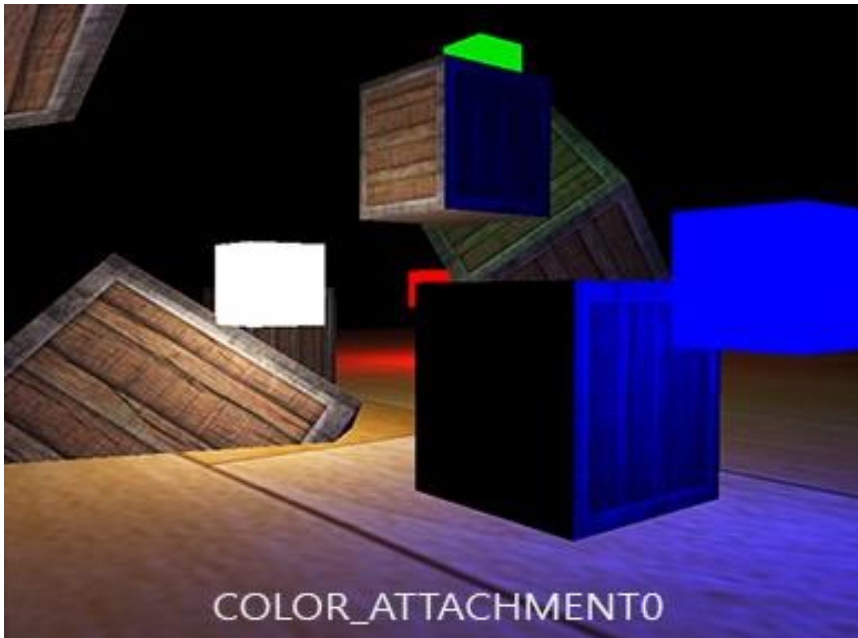
```
#version 330 core
layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;
[...]
void main()
{
    [...] // first do normal lighting calculations and output results
    FragColor = vec4(lighting, 1.0);
    // if fragment output is higher than threshold, output brightness color
    float brightness = dot(FragColor.rgb, vec3(0.2126, 0.7152, 0.0722));
    if(brightness > 1.0)
        BrightColor = vec4(FragColor.rgb, 1.0);
    else
        BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

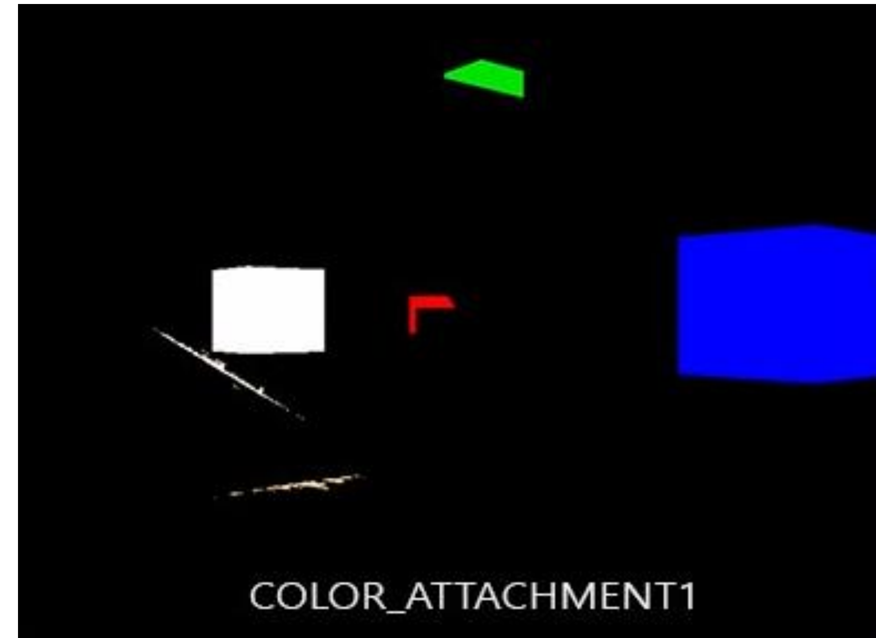
Extracting Bright Color

- This shows why bloom works well with HDR rendering
- Render in HDR, color values can exceed 1.0 → allows to specify a brightness threshold outside the default range
- Without HDR have to set the threshold lower than 1.0 (possible, but regions are much quicker considered as bright)
- Leads to glow effect becoming too dominant (white glowing snow)

Extracting Bright Color

- Two colorbuffers: an image of the scene as normal, and an image of the extracted bright regions; all obtained in a single render pass

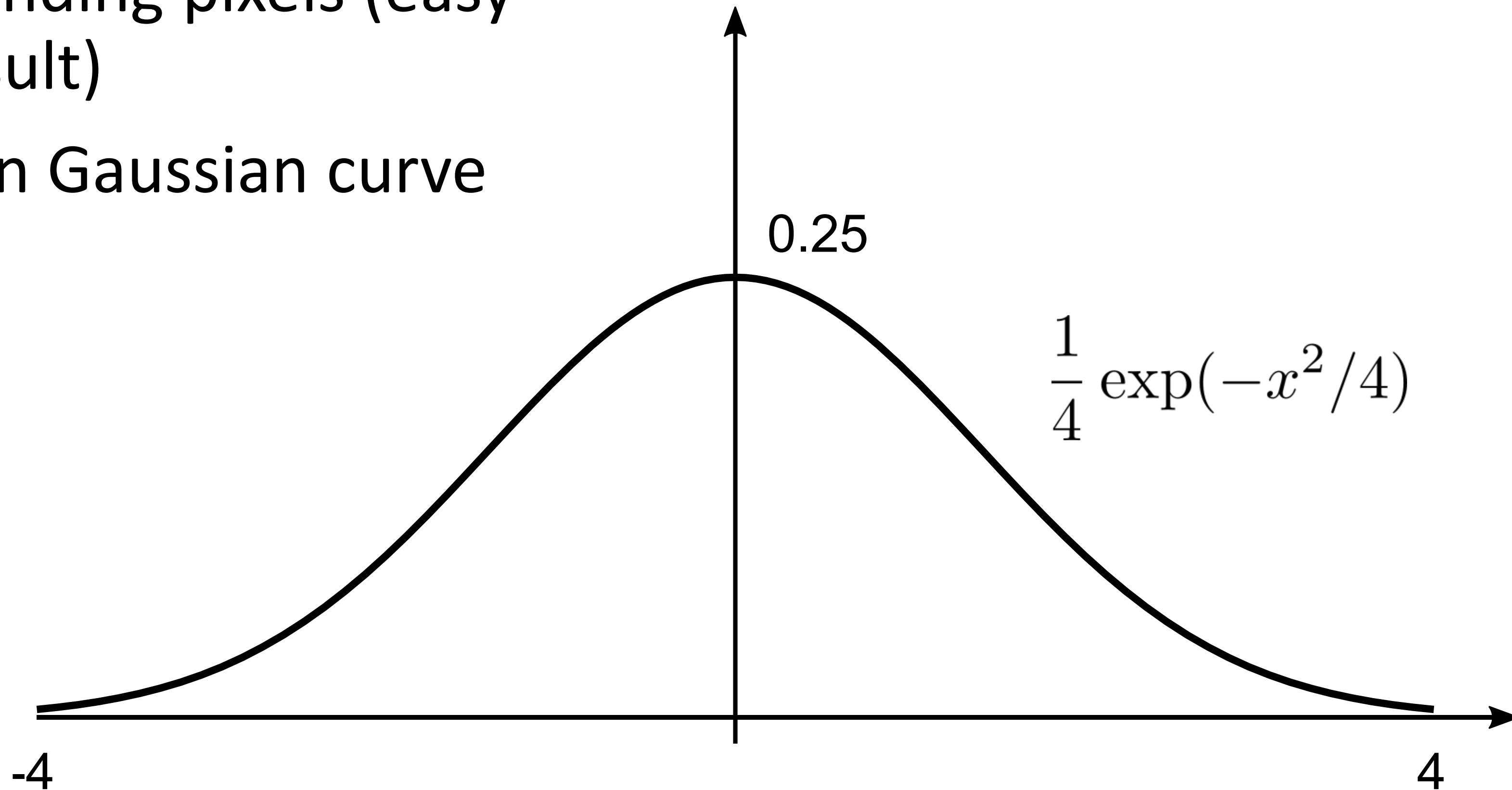




Gaussian Blur

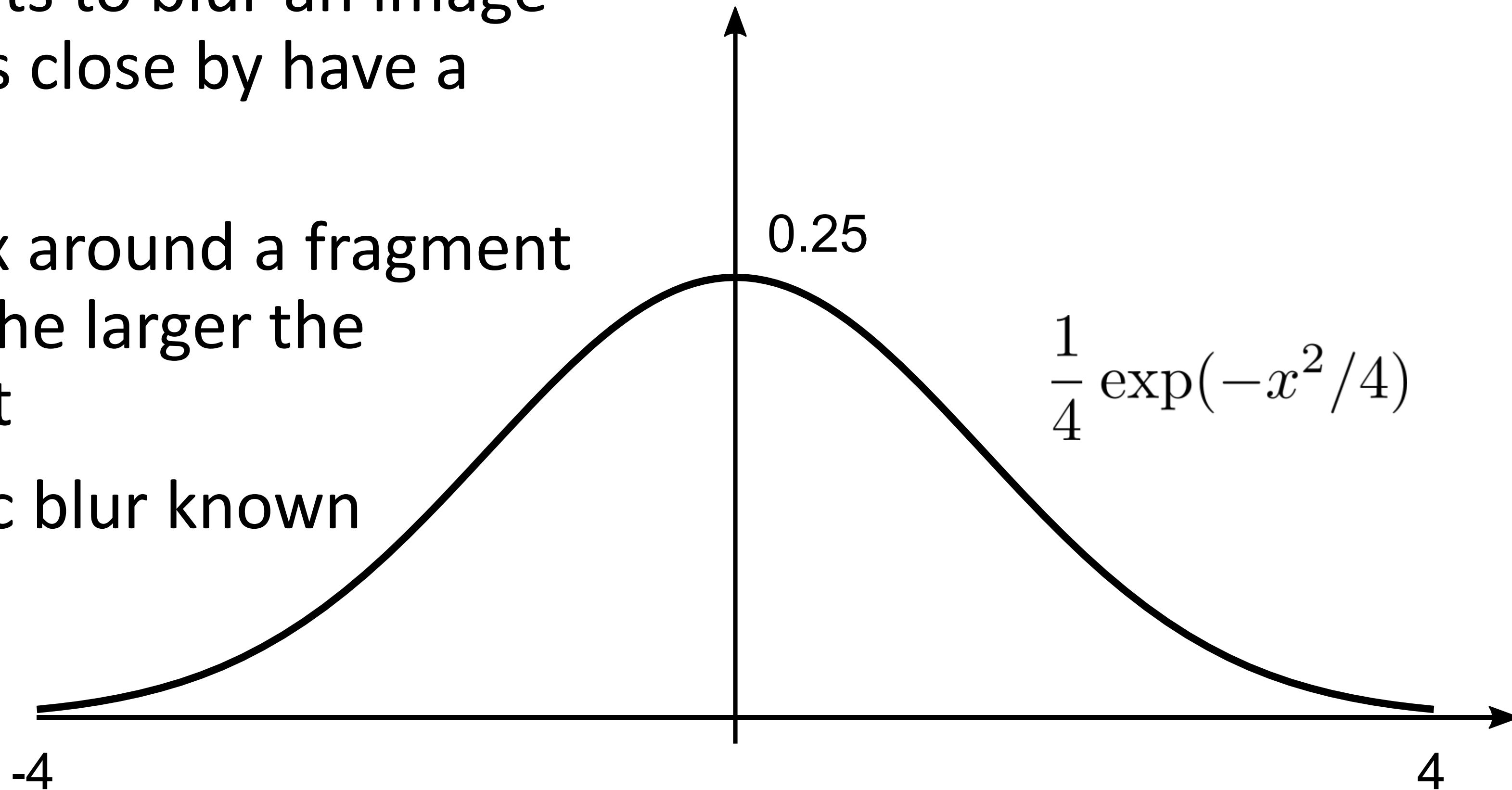
Gaussian Blur

- In the post-processing blur we simply took the average of all surrounding pixels (easy blur but not the best result)
- A Gaussian blur based on Gaussian curve
- Example:



Gaussian Blur

- Gaussian larger area close to its center, using its values as weights to blur an image (better result as samples close by have a higher precedence)
- E.g., sample a 32x32 box around a fragment → use smaller weights the larger the distance to the fragment
- Better and more realistic blur known as a Gaussian blur

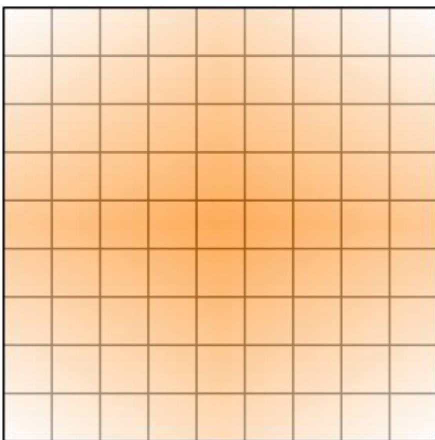


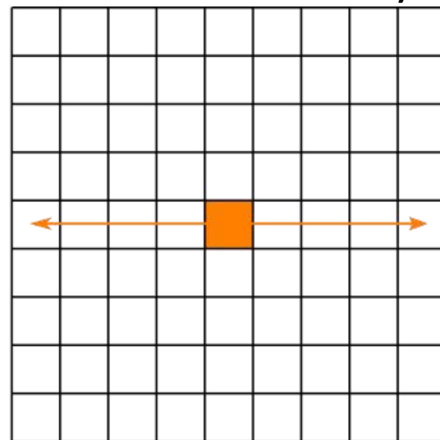
Gaussian Blur

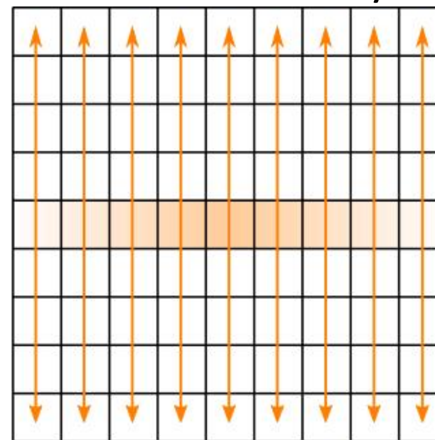
- Implement a Gaussian blur filter need a two-dimensional box of weights → obtain from a 2 dimensional Gaussian curve equation
- Problem is that it becomes extremely heavy on performance
- Take a blur kernel of 32 by 32, this would require to sample a texture a total of 1024 times for each fragment

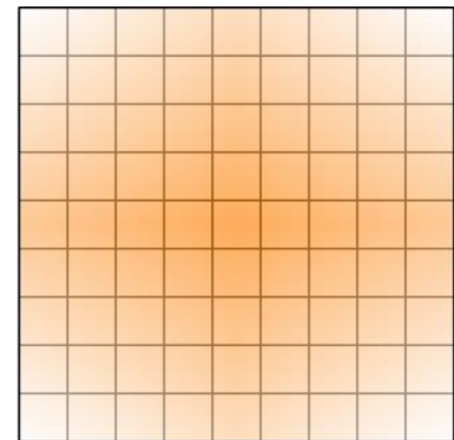
Gaussian Blur

- Gaussian equation property: two dimensional equation separated into two smaller equations: horizontal and vertical weights
- First do a horizontal blur (horizontal weights) then on the resulting texture do a vertical blur
- Results are exactly the same, but saves performance: have to do 32 + 32 samples compared to 1024 (known as two-pass Gaussian blur)


Normal Gaussian Blur


1. Blur Horizontally


2. Blur Vertically


Two-Pass Gaussian Blur

Gaussian Blur

- Means: blur an image at least two times with framebuffer objects
- Implementing a Gaussian blur, need ping-pong framebuffers
- That is a pair of framebuffers, render the other framebuffer’s colorbuffer into current framebuffer’s colorbuffer (with alternating shader effect)
- Switch framebuffer to draw in and also the texture to draw with → first blur the scene’s texture in the first framebuffer, then blur the first framebuffer’s colorbuffer into the second framebuffer and switch

Gaussian Blur

- Gaussian blur's fragment shader:

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;

uniform sampler2D image;

uniform bool horizontal;
float weight[5] = float[] (0.2270270270, 0.1945945946, 0.1216216216, 0.0540540541,  0.0162162162);

void main()
{
     vec2 tex_offset = 1.0 / textureSize(image, 0); // gets size of single texel
     vec3 result = texture(image, TexCoords).rgb * weight[0];
     …
```

Gaussian Blur

- Gaussian blur's fragment shader:

```
…
    if(horizontal)
    {
        for(int i = 1; i < 5; ++i)
        {
           result += texture(image, TexCoords + vec2(tex_offset.x * i, 0.0)).rgb * weight[i];
           result += texture(image, TexCoords - vec2(tex_offset.x * i, 0.0)).rgb * weight[i];
        }
    }
    else
    {
        for(int i = 1; i < 5; ++i)
        {
            result += texture(image, TexCoords + vec2(0.0, tex_offset.y * i)).rgb * weight[i];
            result += texture(image, TexCoords - vec2(0.0, tex_offset.y * i)).rgb * weight[i];
        }
    }
    FragColor = vec4(result, 1.0);
}
```

Gaussian Blur

- Blurring an image, create two framebuffers, each with a colorbuffer texture:

```
unsigned int pingpongFBO[2];
unsigned int pingpongColorbuffers[2];
glGenFramebuffers(2, pingpongFBO);
glGenTextures(2, pingpongColorbuffers);
for (unsigned int i = 0; i < 2; i++)
{
   glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[i]);
   glBindTexture(GL_TEXTURE_2D, pingpongColorbuffers[i]);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);

   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);

   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
         pingpongColorbuffers[i], 0);
}
```

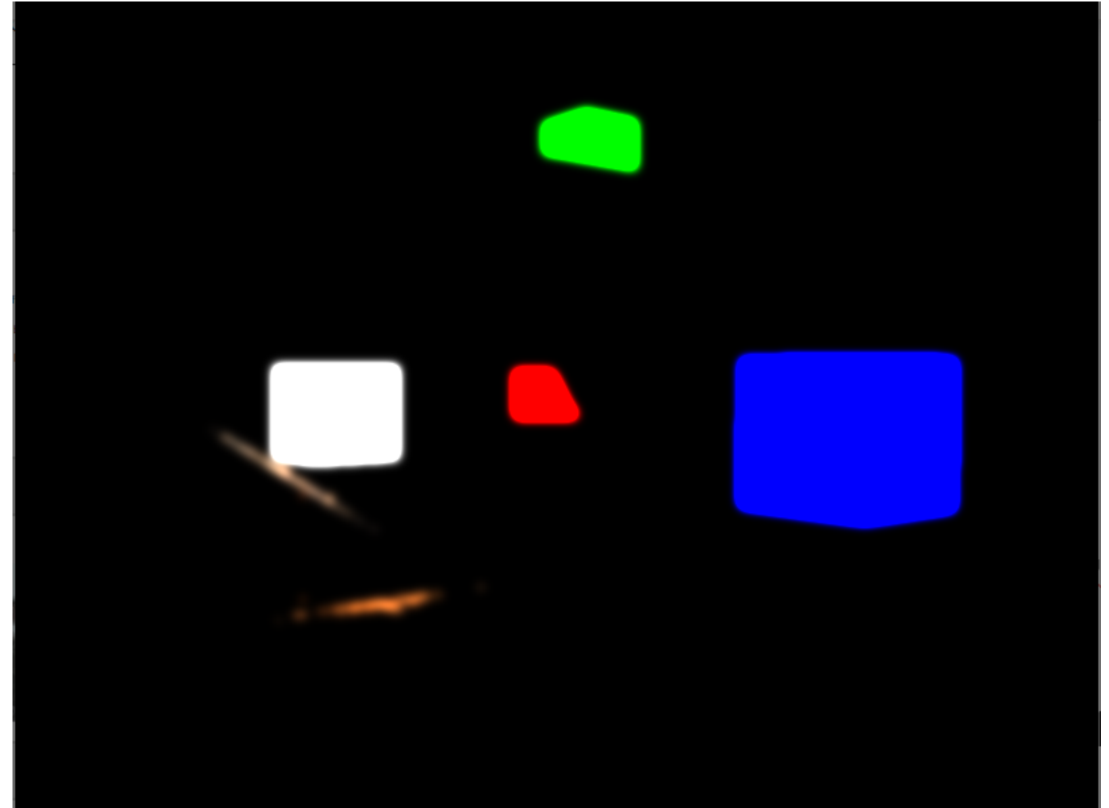
Gaussian Blur

- With an HDR texture, an extracted brightness texture: fill one of the ping-pong framebuffers with the brightness texture and then blur the image 10 times (5 times horizontally and 5 times vertically):

```
bool horizontal = true, first_iteration = true;
unsigned int amount = 10;
shaderBlur.use();
for (unsigned int i = 0; i < amount; i++)
{
   glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[horizontal]);
   shaderBlur.setInt("horizontal", horizontal);
   glBindTexture(GL_TEXTURE_2D, first_iteration ? colorBuffers[1] :
         pingpongColorbuffers[!horizontal]);
   renderQuad();
   horizontal = !horizontal;
   if (first_iteration)
      first_iteration = false;
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Gaussian Blur

- Blurring the extracted brigtness texture 5 times gives a blurred image of all bright regions of a scene



Blending Both Textures

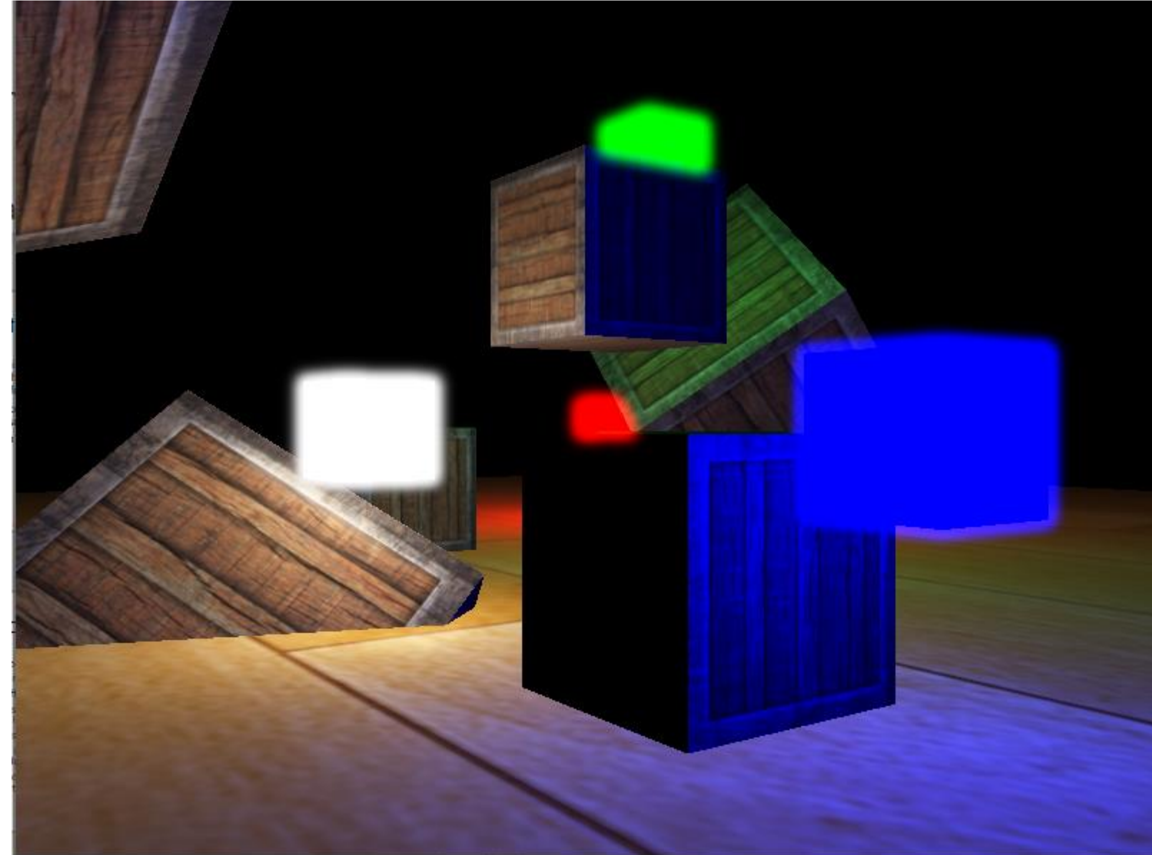
Blending Both Textures

- With HDR, blurred brightness texture, now need to combine them
- In the final fragment shader, blend both textures (HDR lecture):

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D scene;
uniform sampler2D bloomBlur;
uniform bool bloom;
uniform float exposure;
void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(scene, TexCoords).rgb;
    vec3 bloomColor = texture(bloomBlur, TexCoords).rgb;
    if(bloom)
        hdrColor += bloomColor; // additive blending
    // tone mapping
    vec3 result = vec3(1.0) - exp(-hdrColor * exposure);
    // also gamma correct while we're at it
    result = pow(result, vec3(1.0 / gamma));
    FragColor = vec4(result, 1.0);
}
```

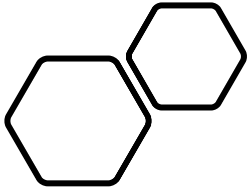
F5…

- … proper glow effect!



Note

- Used a simple Gaussian blur filter (only 5 samples in each direction)
- By taking more samples along a larger radius or repeating the blur filter an extra number of times improves the blur effect
- Quality of the blur directly correlates to the quality of the bloom effect improving the blur step can make a significant improvement
- Some improvements combine blur filters with varying sized blur kernels or multiple Gaussian curves to selectively combine weights



Questions???