DH2323 Report
Bloom Project

Yan Yu
yan8@kth.se

Abstract
This is an implementation of the Bloom effect in OpenGL.

Introduction
Bloom is a post-processing effect in computer graphics, frequently seen in video games and HDR (high dynamic range) rendering. The effect simulates how the camera captures a very bright light in the real world. As a digital simulation, it produces fringes of light extending from the borders of bright areas in an image, creating the illusion of a bright light bleeding over its edge.[1]
The fundamental implementation of Bloom is to filter out the overly bright HDR colors from the main scene, apply downsampling and gaussian blur to the bright layer, and then upsample and add the extra layers to the original scene.[2] This method was first presented by Chris Tchou at Gamefest 2006.[3] Downsampled layers can give a wider blur radius and save computation compared to a full resolution blur buffer.
In this project, OpenGL was used as the program for implementation. The skeleton of the code, including code structure and function initiation, was derived from the Bloom tutorial on learnopengl.com.[4] The scene was customized, and a different Bloom pipeline was used. The main objective of the project is to implement the downsample-and-blur method described above.

Methods
Tone Mapping
HDR imaging refers to the technique that produces a greater dynamic range of luminosity in an image.[5] It allows the display of colors that exceeds the RGB range of [0, 255] (or the normalized range between 0.0 and 1.0 as in OpenGL). Tone mapping is applied to compress the large color values above 1.0 to the 0.0-1.0 range. The method chosen in this project was derived from learnopengl.com. It uses an exponential function to scale down the color value, with an exposure variable to control the overall brightness of the scene.

```
1.  vec3 result = vec3(1.0) - exp(-hdrColor * exposure);
```

Scene and Lighting

[1] Bloom (shader effect). Wikipedia.
[2] Philip Rideout. OpenGL Bloom Tutorial. https://prideout.net/blog/old/archive/bloom/index.php.html
[3] Compute Shader HDR and Bloom. 2012. Intel. https://www.intel.com/content/www/us/en/developer/articles/code-sample/compute-shader-hdr-and-bloom.html?wapkw=bloom
[4] Bloom. Learn OpenGL. https://learnopengl.com/Advanced-Lighting/Bloom
[5] High Dynamic Range (HDR). OpenCV. https://docs.opencv.org/3.4/d2/df0/tutorial_py_hdr.html

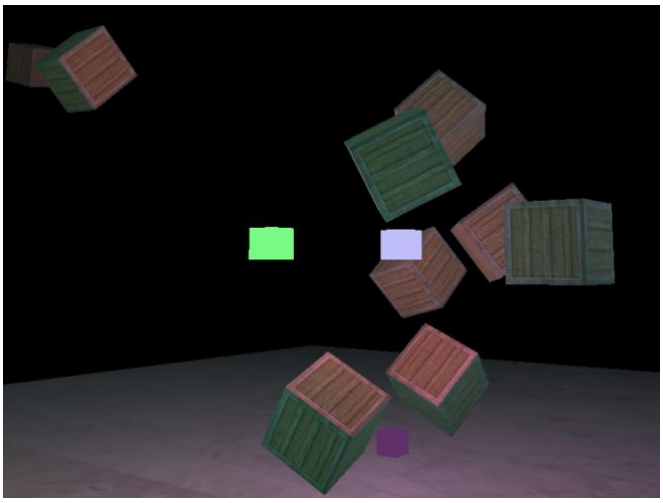

Fig. 1 Basic Scene

A basic scene was set up with cubes and a plane, with the colored cubes representing point light sources. The two light cubes in the middle have high RGB values well over 1.0. The fragment shader calculates point light reflections through ambient, diffuse and specular shading. The perceived bright colors are differentiated by the luminance formula below, and then rendered to another color texture.

```
1.  float brightness = dot(FragColor.rgb, vec3(0.2126, 0.7152, 0.0722));
2.  if(brightness > 1.0f)
3.      BrightColor = vec4(FragColor.rgb, 1.0);
4.  else
5.      BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
```

Framebuffers

A framebuffer is a collection of buffers used as the destination for rendering.[6] For this project, multiple framebuffers are needed for the layering of color textures. After the window, vertex objects and shaders were initiated, the main scene and the bright layer were rendered as two separate color textures on an FBO (framebuffer object). Each downsampled and gaussian blurred texture also requires a new FBO to render.

Downsampling

The bright color texture was downsampled four times in order to get larger contours of the light cube before blurring. The downsampling passes were achieved using glViewport, which resized the window with 1/2, 1/4, 1/8 and 1/16 of the screen height and width. GL_LINEAR function was used to set up the color textures so that the pixels in the upsampled textures would be blurred. The code below shows the rendering process for the four textures.

```
1.  quadShader.use();               // shader to render a screen quad
```

[6] Framebuffer. OpenGL Wiki. https://www.khronos.org/opengl/wiki/Framebuffer

```
for (int i=0; i<num_layers; i++) {     // num_layers = 4
    glBindFramebuffer(GL_FRAMEBUFFER, downsampleFBO[i]);
    int fraction = pow(2, i+1);
    glViewport(0, 0, int(SCR_WIDTH / fraction), int(SCR_HEIGHT / fraction) );
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, colorBuffers[1]); // bind the bright texture
    renderQuad(quadVAO);             // render the downsampled textures
}
```

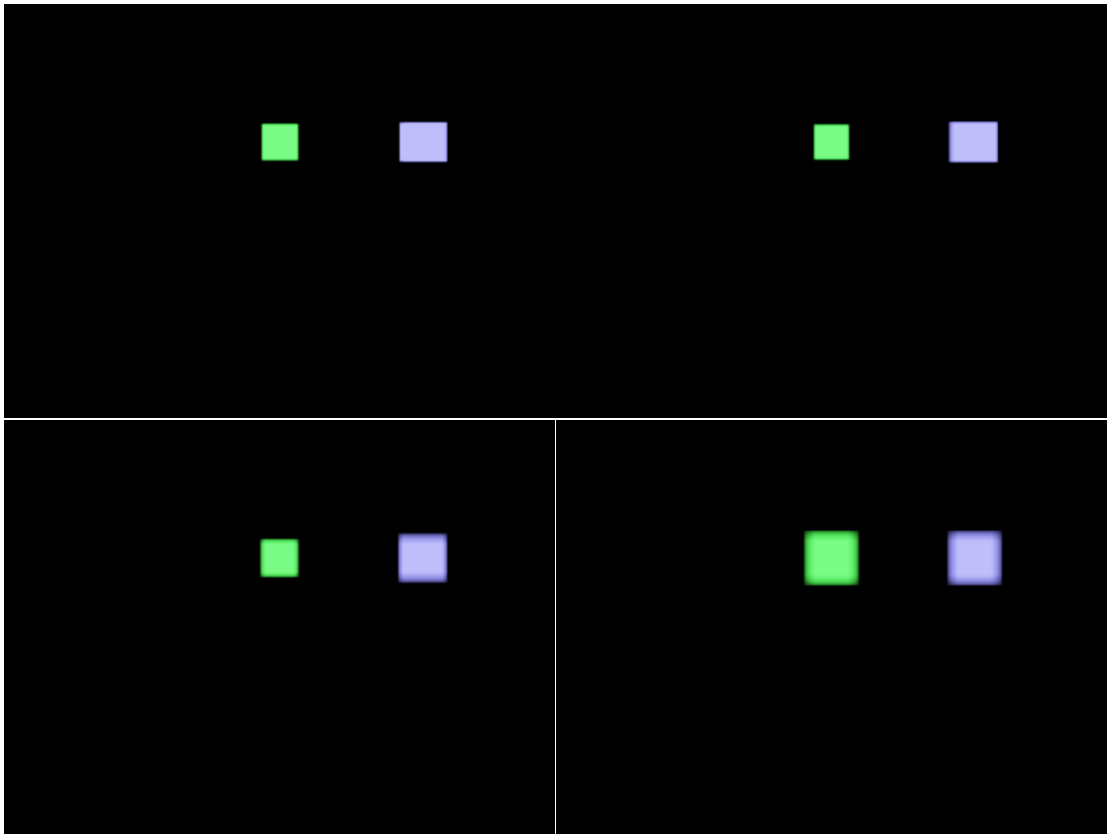

Fig 2. The bright layer downsampled with 1/2x1/2, 1/4x1/4, 1/8x1/8 and 1/16x1/16 sizing

Gaussian Blur

After downsampling, a Gaussian blur of 5-pixel radius was applied to each of the four textures. In order to save computational power, the blur shader is set up to separate the horizontal pass from the vertical pass. For each downsampled texture, a pair of FBOs (also called ping pong FBOs) were created. The shader could then be repeatedly applied to the two buffers back-and-force to achieve multiple blur iterations. The equation for calculating the gaussian weights and the blurred results are shown below.[7]

[7] Gaussian blur. Wikipedia. https://en.wikipedia.org/wiki/Gaussian_blur

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

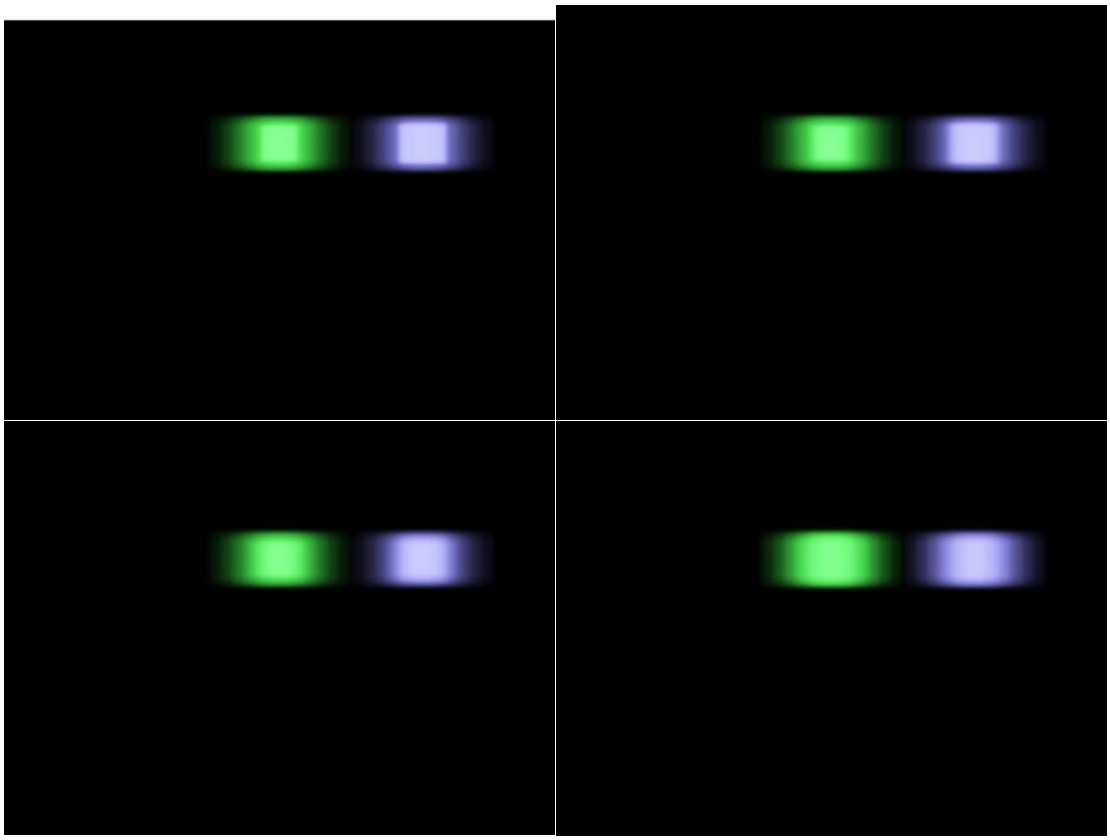

Fig 3. The four downsampled textures after gaussian blur

Results

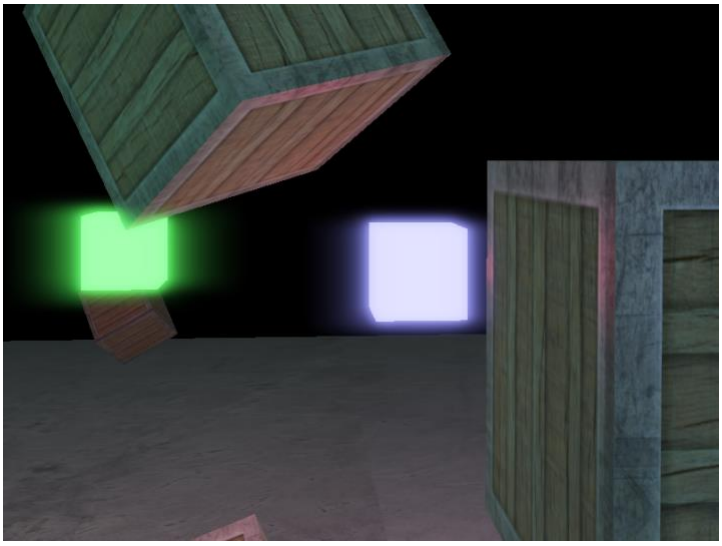

Fig 4. Final Bloom effect

The image above shows the bloom effect after the original texture and the four blurred textures were added up together. Each downsampled texture passed through 10 rounds of gaussian blurs. Keys were mapped to the brightness threshold, the exposure value and the spread of the gaussian blur to allow for more detailed controls.

Reflections

As I was unfamiliar with OpenGL before the project, it took me a while to learn the graphics pipeline and the basic functions in OpenGL. I also encountered a lot of issues with framebuffers which took a significant amount of time to fix. There was the issue of the blurred textures not aligning with the original texture, a problem I could only fix by rendering to full screen. The way how the framebuffers were bound was also prone to error, and very likely to produce a black screen. It makes the program very challenging to debug, but throughout the process I become more experienced with the graphics pipeline and graphics programming.

Performance-wise, the effect above would require a lot of GPU power because of the extensive rounds of gaussian blurs. The image below shows a more performance-friendly version with fewer rounds of gaussian blurs, however with trade-off on the Bloom quality—the outer fringes of the cubes are not smooth enough and the overall shape looks blurred.

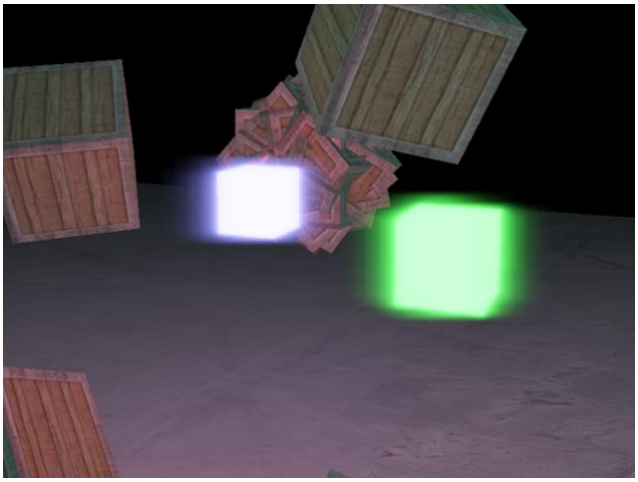

Fig 4. The effect with fewer gaussian blurs

This project could very well be improved by a user study on the perception of bloom effect. The effect in different settings, such as exposure, blur radius, number of downsample passes, etc., could be presented to a user group for comparison. It would provide evidence of which parameters are more relevant to the perception of Bloom effect. Additionally, a survey could be conducted consisting of this custom effect and professionally made Bloom effects. A similar scene could be set up with different Bloom algorithms in effect. The users would then be able to give feedback on what aspects of this implementation are lacking compared to the Bloom effects in games and 3D engines. Through such study I would have a better idea of how to further improve the implementation.

There are also things I wish I could do more with this effect. Due to the trouble with setting up the assimp library, I couldn't test this effect on models with more complicated shapes. It would be

interesting to see how the effect shows in a more detailed and “realistic” scene. I intend to further develop this effect in the future and use it in more practical work such as game development or computer animation.

Related Work

1. Philip Rideout. OpenGL Bloom Tutorial. https://prideout.net/blog/old/archive/bloom/index.php.html
2. The Cherno. Bloom. 2021. Youtube. https://www.youtube.com/watch?v=tI70-HIc5ro&t=3s
3. Compute Shader HDR and Bloom. 2012. Intel. https://www.intel.com/content/www/us/en/developer/articles/code-sample/compute-shader-hdr-and-bloom.html?wapkw=bloom
4. Bloom. Learn OpenGL. https://learnopengl.com/Advanced-Lighting/Bloom