JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 26 No. 2 (2018), pp. 31-43

Heterogeneous Fog Generated with the Effect of Light Scattering and Blur

Michał Gawron[1], Urszula Boryczka[2]

[1]*University of Silesia*
Institute of Computer Science
Będzińska 39, 41-200 Sosnowiec, Poland
michalgawron.mg@gmail.com

[2]*University of Silesia*
Institute of Computer Science
Będzińska 39, 41-200 Sosnowiec, Poland
urszula.boryczka@us.edu.pl

Abstract. *The development of computer graphics forces new requirements on the developers, which will make the virtual world more similar to the real world. One of these elements is the simulation of fog. Common fog algorithms mix the color of the scene with the color of the fog over a certain distance. However, one feature of the naturally foggy scenery is ignored. With the distance and density of the fog, the observed scenery or individual objects become more blurred. In this paper we will present our implementation of the distance fog in the Unreal Engine 4, including the effect of blurring the foggy areas, simulating of light scattering and variations in fog density using noise.*

Keywords: *computer graphics, computer games, fog, blur, Unreal Engine.*

1. Introduction

Visualization of natural phenomena is an important part of the current development process of computer games. The development of computer graphics imposes new requirements on the developers, which will make the virtual world even more similar to the real world. One of these phenomena is the simulation of fog, which in games is used primarily to create an atmosphere around the player. Over the years, the way it is rendered has changed many times to express its natural properties. Currently, the most advanced method is volumetric fog, which provides the highest quality and at the same time high calculation cost.

Common fog algorithms mix the color of the scene with the color of the fog over a certain distance. However, one feature of the naturally foggy scenery is ignored. With the distance and density of the fog, the viewed scenery or individual objects become more blurred. This feature can be seen in the figure 1, where the Westminster Palace and the Elizabeth Tower seen in the distance are blurred, which makes them lacking in detail. A noticeable contrast are the clear and sharp light lamps in the foreground.

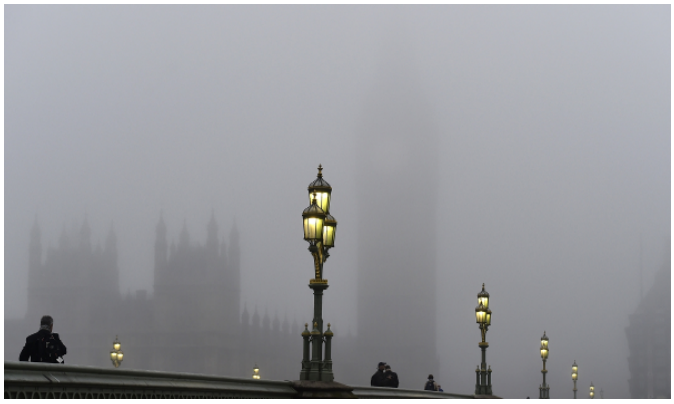

Figure 1: Westminster Palace on a foggy day. Source: https://www.pri.org

In this paper we will present our implementation of the real-time distance fog in the Unreal Engine 4. The post-process effect includes the blurring effect described earlier, simulation of light scattering and variations in fog density using noise.

2. Related work

When adding the fog to the game there is a large number of algorithms that can be implemented. The simplest implementations include a uniform fog that mixes scene color with fog color based on the pixel distance from the observer. The algorithms described at [1] and [2] use noise to simulate the feeling of heterogeneity and the movement of suspended in-air water particles. The effect of light scattering in fog is described in [3] [4] [5] and [6]. The latest and most advanced implementations are based on volumetric solutions that fully express the feeling of fog density and the light scattering [7] and [8].

3. Proposed algorithm

Our fog implementation proposed in the paper in Unreal Engine 4 is based on four elements:

- generating distance fog;
- simulating light scattering;
- simulating heterogeneity and motion with noise;
- generating scene blur depending on fog density.

3.1. Generating distance fog

Based on the given parameters determining the distance, density and color, we generate a fog which is then mixed with the scene. The easiest way to calculate the fog distance is to calculate it using the depth buffer. However, it causes some distortion - the fog is calculated linearly from the camera. For this reason, the distance is calculated on the basis of the position of the camera and the world position reconstructed from the depth buffer. This creates fog in the circle from the camera (fig. 2).

The fog density is calculated from formula 3 based on the parameters determining the distance and density. Then the color of the scene is blended with the fog color using equation 4.

$$\overrightarrow{p_d} = |WorldPos - CameraPos|; \tag{1}$$

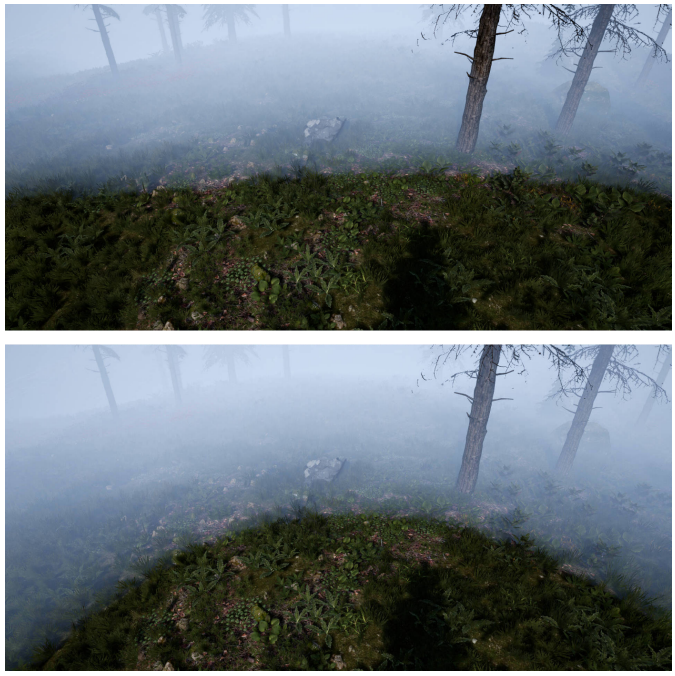

Figure 2: Comparison of fog generation using depth buffer (top) and world/camera position (bottom)

$$dist = \left(\frac{|\overrightarrow{p_d}| - StartDistance}{Distance} \right)^{Exponent} \tag{2}$$

$$F_{Density} = max(dist, FogDensity) \tag{3}$$

$$C_{Final} = C\left(1 - F_{Density}\right) + C_{Fog}F_{Density} \tag{4}$$

where $WorldPos$ is the scene world position reconstructed from depth buffer, $CameraPos$ is the camera world position, $StartDistance$ is the fog offset start distance, $Distance$ is the fog blend distance, $Exponent$ is the fog blend distance falloff, $FogDensity$ is the max value of fog density, C is the scene base color and C_{Fog} is the fog color.

3.2. Simulating light scattering

The simulation of light scattering by fog is possible when the camera is directed towards a light source, in most cases it is a directional light. This effect is achieved by calculating the angle between the incident light and the camera. Based on the calculated angle, the color responsible for the scattering is added to the base color of the fog. Figure 3 shows an example of how a soft orange scattering from a light source is added to the grey-blue fog.

$$\overrightarrow{p_s} = (WorldPos - CameraPos) + ScatterStartDist \tag{5}$$

$$scatter = \frac{\overrightarrow{p_s}}{|\overrightarrow{p_s}|} \circ \overrightarrow{L} \tag{6}$$

$$F_{Scatter} = (max(0, scatter))^{ScatterExponent} \tag{7}$$

where $WorldPos$ is the scene world position reconstructed from depth buffer, $CameraPos$ is the camera world position, $\overrightarrow{L}$ is the light vector, $ScatterStartDist$ is the fog scatter offset distance and $ScatterExponent$ is the fog scatter falloff.

With the calculated coefficient of scattering 7 we mix the fog color and the scatter color using the formula below:

$$C_{Fog} = C_{Fog} + C_{Scatter} F_{Scatter} \tag{8}$$

where C_{Fog} is the fog color and $C_{Scatter}$ is the fog scatter color.

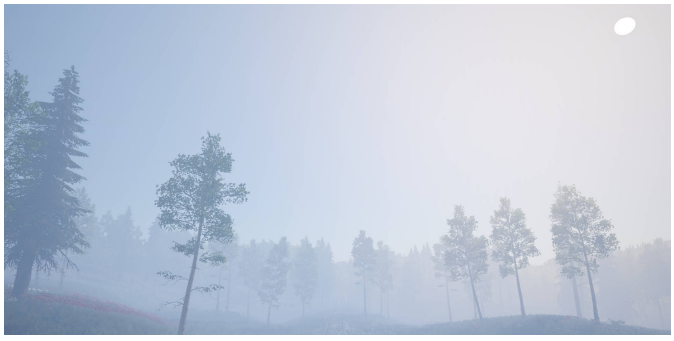

Figure 3: Fog light scattering effect

3.3. Fog density variation

The fog generated in the previous stages has a uniform density and color all around the area. To achieve variation, we used noise texture with the world position as its coordinates. Real-time noise generation is a very complex process with very low performance. That's why we decided to use the previously generated texture, which we then sample in the shader. The feeling of motion is created by shifting the noise coordinates in time by a given value specified by the user. The noise generated in this way is used to manipulate the density and color of fog on the scene (fig. 4). It is possible to modify the fog density using black and white noise texture based on the formula 9 and insert it into the equation 3.

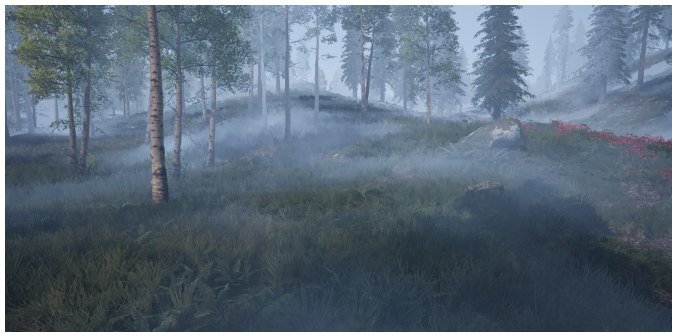

Figure 4: Example variation of the fog density

$$N_{Density} = Low_D\,(1 - Noise) + Max_D\,Noise \tag{9}$$

$$F_{Density} = max(N_{Density}\,dist, FogDensity) \tag{10}$$

where $Noise$ is the noise texture, Low_D is the lowest fog density, Max_D is the highest fog density and $FogDensity$ is the global max value of fog density.

Listing 1 shows the complete HLSL shader code used to generate the fog, including examples of parameter values. Figure 5 shows a comparison of the default fog from the Unreal Engine 4 and our implementation. As one can see, they are very similar to each other.

Listing 1: Shader code of fog generation

```
1 // example parameters setup
```

```
2 float StartDistance = 0.0f;
3 float Distance = 1000.0f;
4 float Exponent = 0.55f;
5 float3 NoiseSpeed = float3(200.0f, 25.0f, 0.0f);
6 float NoiseSize = 10000.0f;
7 float LowNoiseDensity = 0.12f;
8 float MaxNoiseDensity = 0.18f;
9 float MaxDensity = 0.5f;
10 float ScatterStartDist = 1000.0f;
11 float ScatterExponent = 4.0f;
12 float3 FogColor = float3(0.45f, 0.64f, 1.0f);
13 float3 ScatterColor = float3(1.0f, 0.65f, 0.28f);
14
15 float FogDistance(float Offset, float Distance, float Exponent)
16 {
17    float position = WorldPosition - CameraPosition;
18    float dist = sqrt(dot(position, position));
19    return pow((dist + Offset) / Distance, Exponent);
20 }
21
22 float NoiseDensity(Texture NoiseTex, float3 NoiseSpeed, float NoiseSize)
23 {
24    float3 uv = WorldPosition + NoiseSpeed * Time;
25    uv /= NoiseSize;
26    return tex2D(NoiseTex, uv.xy).x;
27 }
28
29 float FogScatter(float Offset, float Exponent)
30 {
31    float position = (WorldPosition - CameraPosition) + Offset;
32    position = normalize(position);
33    float dist = saturate(dot(position, LightVector));
34    return pow(dist, Exponent);
35 }
36
37 float3 ComputeFog()
38 {
39    float density = FogDistance(StartDistance, Distance, Exponent);
40    float noise = NoiseDensity(NoiseTex, NoiseSpeed, NoiseSize);
41    density *= lerp(LowNoiseDensity, MaxNoiseDensity, noise);
42    float finalDensity = clamp(0.0f, MaxDensity, density);
43
44    float scatter = FogScatter(ScatterStartDist, ScatterExponent);
45    float3 finalFog = FogColor + scatter * ScatterColor;
46
```

```
47     return lerp(SceneColor, finalFog, finalDensity);
48 }
```

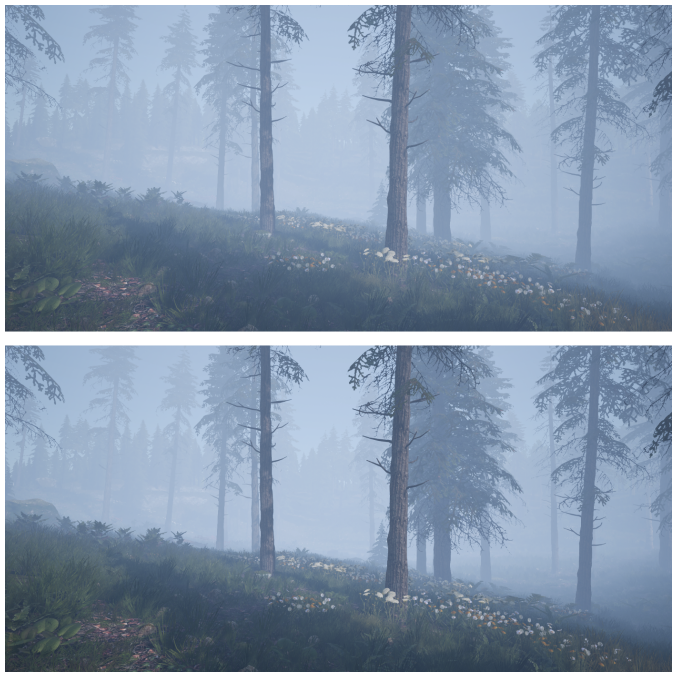

Figure 5: Comparison of the default Unreal Engine 4 fog (top) and our implementation (bottom)

3.4. Generating scene blur

The generation of blur is the main point of our fog implementation. There are many blur algorithms. Initially we used a fast horizontal-vertical algorithm (box blur). However, the result quality was insufficient and for that reason we used a Gaussian blur. This allows us to achieve high image quality even at high blur values. Figures 6 and 7 shows four examples where blurring increases the realism of a scene.

The Gaussian blur is a type of image-blurring filter that uses a Gaussian function for calculating the transformation to apply to each pixel in the image [9] [10]. The equation of a Gaussian function in two dimensions is:

Table 1: Kernel weights matrix for Gaussian blur

0.002915	0.013064	0.021539	0.013064	0.002915
0.013064	0.058550	0.096532	0.058550	0.013064
0.021539	0.096532	0.159155	0.096532	0.021539
0.013064	0.058550	0.096532	0.058550	0.013064
0.002915	0.013064	0.021539	0.013064	0.002915

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{11}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis and σ is the standard deviation of the Gaussian distribution.

Values from distribution 11 are used to build a convolution matrix which is applied to the original image. In our case, we used standard deviation $\sigma = 1.0$ and the kernel size $n = 5$ to create a weights matrix 1.

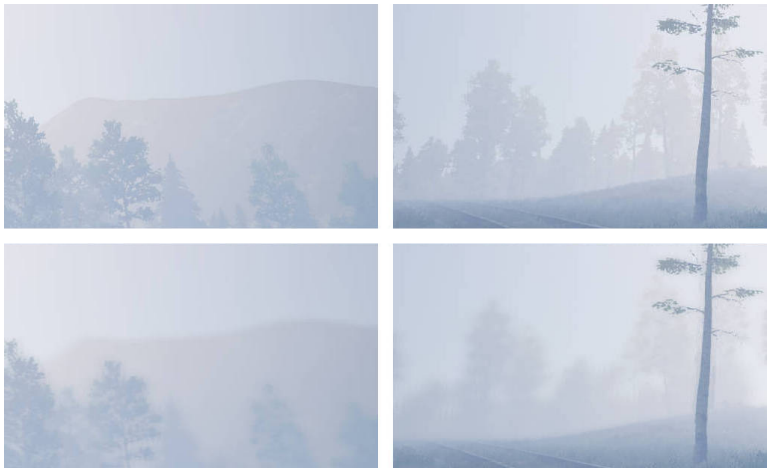

Figure 6: Comparison of example scenes without and with blurring

There are two ways to implement blurring. The first is to generate blur in single pass together with fog generation. Then the blur is created on the basis of an

image without fog. The second solution is a new, separate pass in which the blur is created on an already foggy scene. Implementation differences are shown in the figure 8. No matter what approach we choose, the user can change the blurring parameters, including: distance, falloff or maximum blur value. The blur is related to the density of the fog, and when the density is higher, the blurring value of the scene is also increased.

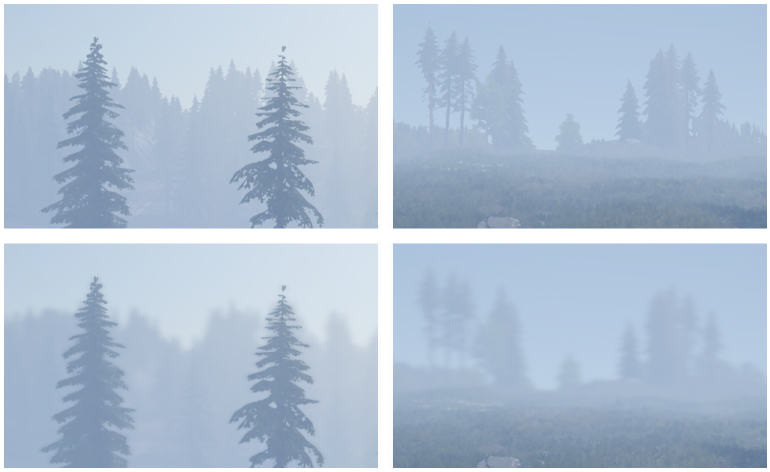

Figure 7: Comparison of example scenes without and with blurring

4. Performance

We have prepared four test scenes to measure the performance of the proposed fog generating solution. Each of the scenes presents a forest landscape. We added our fog algorithm and default Unreal Engine 4 fog to each of them. Then the performance was measured using the GPU profiler tool. We used a computer equipped with Intel Core i5 3.20GHz CPU, NVIDIA GeForce GTX 970 GPU and 16GB of RAM memory. Each scene was started 20 times with the highest quality settings and Full HD resolution (1920x1080).

The results of the performance tests are presented in Tables 2 and 3. As can be seen, our fog solution is only 0.02 ms slower without blurring. Adding the blur as a separate pass increases the rendering time to 0.65ms. However, the combination of fog generation and blurring in a single pass gives better performance results -

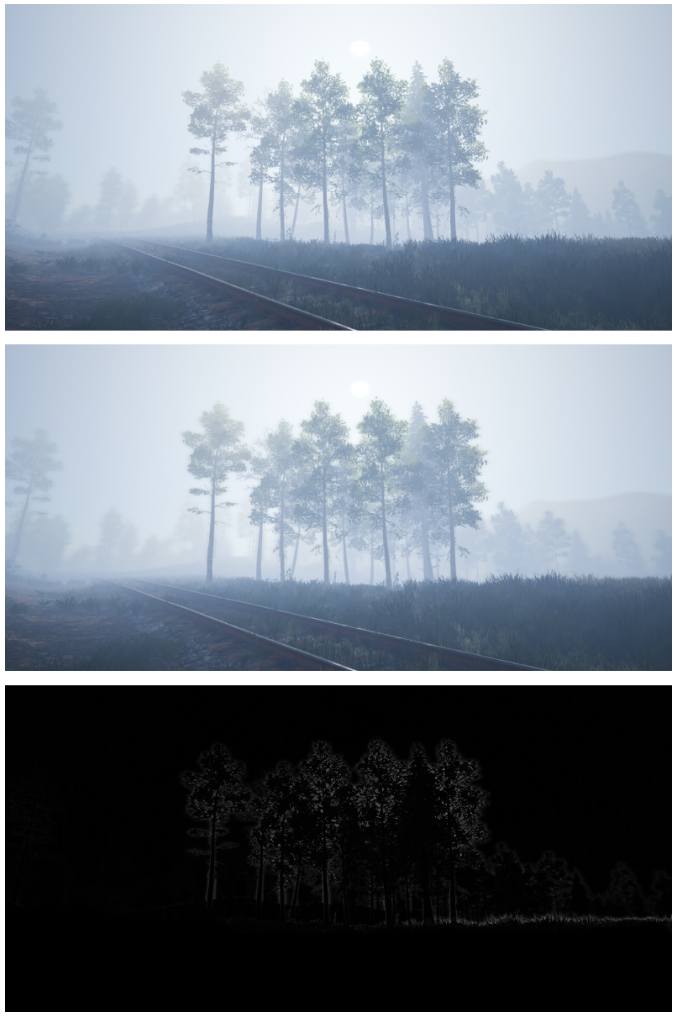

Figure 8: An example of a scene with fog and blur in single pass (top), in separate passes (middle) and the difference between them (bottom)

rendering time is reduced to 0.44ms.

Table 2: Fog performance without blur (in ms)

Scene	**Our implementation**	**Defualt UE4**
Test 01	0.19	0.18
Test 02	0.21	0.16
Test 03	0.20	0.19
Test 04	0.20	0.17
Average	0.20	0.18

5. Conclusion

The fog generated with the use of the proposed solution allows you to quickly add it to any scene. Thanks to the application of blurring of distant objects on the stage, we have achieved a very convincing and close to realism effect. The set of parameters allows us to adjust the fog in any way we like. The way it is made also allows for comfortable modification.

Table 3: Fog performance with blur (in ms)

Scene	**Single pass**	**Separate passes**
Test 01	0.42	0.62
Test 02	0.49	0.71
Test 03	0.41	0.61
Test 04	0.44	0.65
Average	0.44	0.65

A very positive aspect is the rendering time of the fog shown in the work. For four test scenes, the render time of the only fog is 0.20 ms, while adding blur increases the render time to 0.65 ms. The implementation in the form of a full-screen post process effect is like a fog contained directly in the Unreal Engine 4, but this algorithm provides easier management of parameters between scenes, e.g. inside and outside. Our fog algorithm also supports the simulation of motion and heterogeneity of fog density.

Further work on the described fog effect will be focused on improving the generation of blur and taking into account the bloom effect for very bright objects.

References

[1] Giroud, A. and Biri, V., *Modeling and rendering heterogeneous fog in real-time using B-Spline wavelets*, WSCG 2010, 2010, pp. 145–152.

[2] Zdrojewska, D., *Real time rendering of heterogenous fog based on the graphics hardware acceleration*, http://old.cescg.org/CESCG-2004/web/Zdrojewska-Dorota/, 2004, Accessed: 18 June 2018.

[3] Guo, F., Tang, J., and Xiao, X., *Foggy Scene Rendering Based on Transmission Map Estimation*, International Journal of Computer Games Technology, 2014.

[4] Hoffman, N. and Preetham, A. J., *Rendering Outdoor Light Scattering in Real Time*, http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/ATI-LightScattering.pdf, 2002, Accessed: 18 June 2018.

[5] Narasimhan, S. G. and Nayar, S. K., *Shedding Light on the Weather*, In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Vol. I, 2003, pp. 665–672.

[6] Sun, B., Ramamoorthi, R., Narasimhan, S. G., and Nayar, S. K., *A Practical Analytic Single Scattering Model for Real Time Rendering*, ACM Transactions on Graphics, Vol. 24, 2005, pp. 1040–1049.

[7] Billeter, M., Sintorn, E., and Assarsson, U., *Real-time multiple scattering using light propagation volumes*, In: I3D '12 Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2012, pp. 119–126.

[8] Klehm, O., Seidel, H., and Eisemann, E., *Prefiltered single scattering*, In: I3D '14 Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2014, pp. 71–78.

[9] Nixon, M. and Aguado, A. S., *Feature Extraction & Image Processing*, chap. Basic image processing operations, Academic Press, 2nd ed., 2008, pp. 88–90.

[10] Shapiro, L. G. and Stockman, G., *Computer Vision*, chap. Gaussian Filtering and LOG Edge Detection, Prentice Hall, 2001, pp. 166–170.