

Depth of Field

WebGL and Three.js implementations

Marco Benelli Luca Bindini

Università degli Studi di Firenze

July 2021

Table of Contents

1 Introduction

2 WebGL

3 Three.js

4 Conclusion

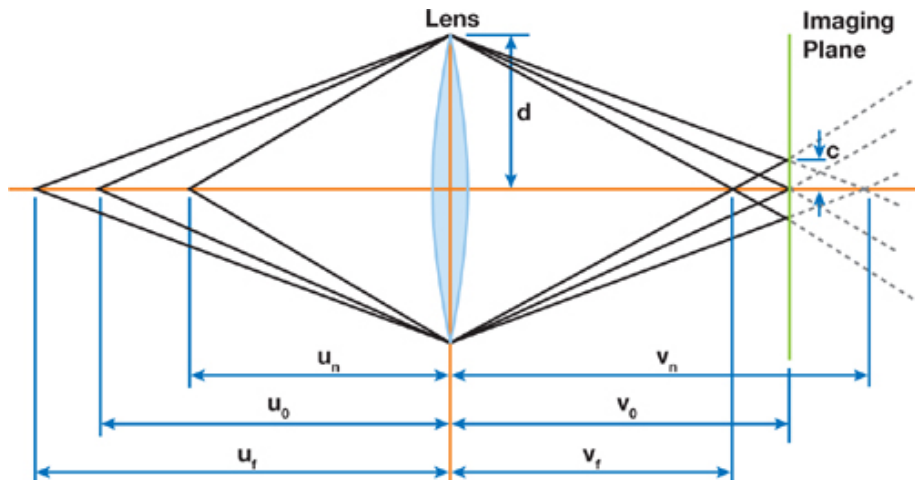
Definition of depth of field

Depth of field is the effect in which objects at a certain distance in a scene appear in focus, and objects nearer or farther appear out of focus.

Some quantities to be familiar with are:

- Focus Length
- Lens Aperture
- Object Distance
- Circle of Confusion

Circle of Confusion



$$c = d \cdot \left| \frac{v_0 - v_f}{v_f} \right| = d \cdot \left| \frac{v_0}{v_f} - 1 \right|$$

How the blur is created

To blur a given pixel we need to give it the color of the average of its neighboring pixels' colors. The bigger the neighborhood, the higher the blurring effect, and the size is given by the circle of confusion.

Three approaches for getting the average are:

- uniform square blur
- uniform circle blur (used by the WebGL implementation)
- gaussian blur (used by the three.js implementation)

Portion of code of vertex shader

```
in vec4 a_Position;
in vec2 a_TexCoord;
uniform mat4 u_ModelViewMatrix, u_ProjMatrix;
uniform float u_d, u_vf;
uniform int u_MaxBlur;
out vec2 v_TexCoord;
out vec4 v_Position;
flat out int v_c;
void main() {
    v_Position = u_ModelViewMatrix * a_Position;
    gl_Position = u_ProjMatrix * v_Position;
    v_TexCoord = a_TexCoord;
    float vo = length(vec3(v_Position));
    v_c = min(int(u_d * abs(vo / u_d - 1.0)), u_MaxBlur);
}
```

Portion of code of fragment shader

```
uniform sampler2D u_Sampler; uniform vec2 u_TexResolution;
in vec2 v_TexCoord; in vec4 v_Position; flat in int v_c;
out vec4 fragColor;
void main() {
    vec3 color = vec3(0.0);
    int total = 0;
    for (int i= -v_c; i <= v_c; ++i)
        for (int j= -v_c; j <= v_c; ++j)
            if (i*i + j*j <= v_c*v_c) {
                color += texture(u_Sampler, v_TexCoord +
                                vec2(i, j) / u_TexResolution).rgb;
                total++;
            }
    fragColor = vec4(color / float(total), 1.0);
}
```

GUI

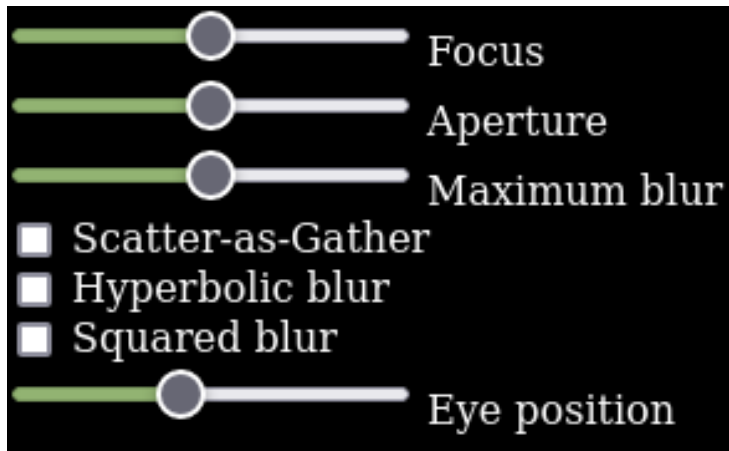
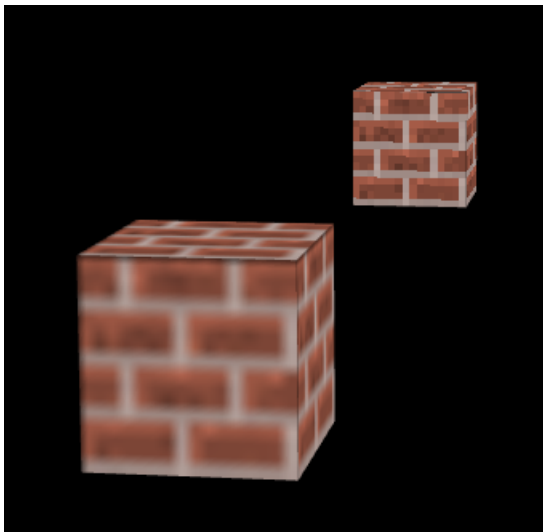
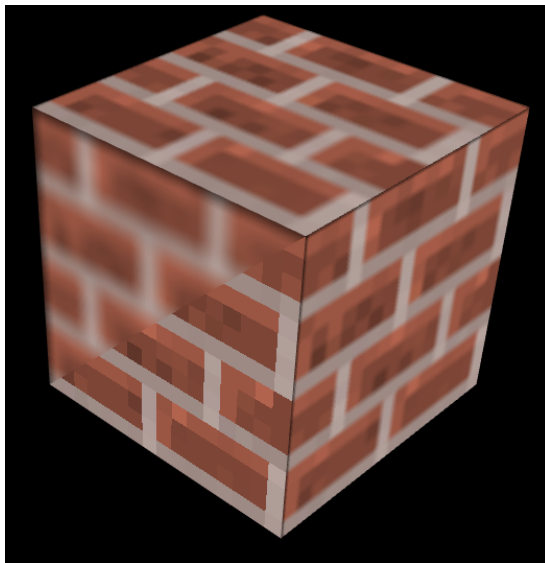


Figure: WebGL graphical user interface

The WebGL scene



Artifacts of the initial stochastic approach



Enhanced version (Scatter as Gather)

Our first algorithm did a gather from neighboring pixels, essentially assuming that each neighbor had a CoC that was the same as the current one. Logically, we would prefer each pixel to smear itself over its neighbors based on its own circle of confusion.

Portion of code of vertex shader (Scatter as Gather)

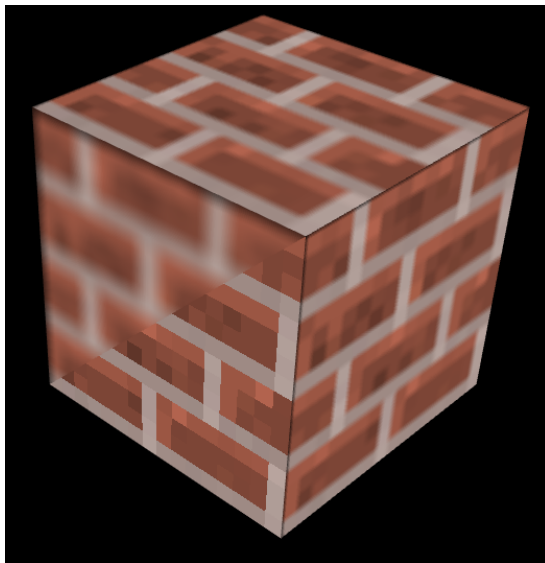
```
in vec4 a_Position;
in vec2 a_TexCoord;
uniform mat4 u_ModelViewMatrix;
uniform mat4 u_ProjMatrix;
out vec2 v_TexCoord;
out vec4 v_Position;

void main() {
    v_Position = u_ModelViewMatrix * a_Position;
    gl_Position = u_ProjMatrix * v_Position;
    v_TexCoord = a_TexCoord;
}
```

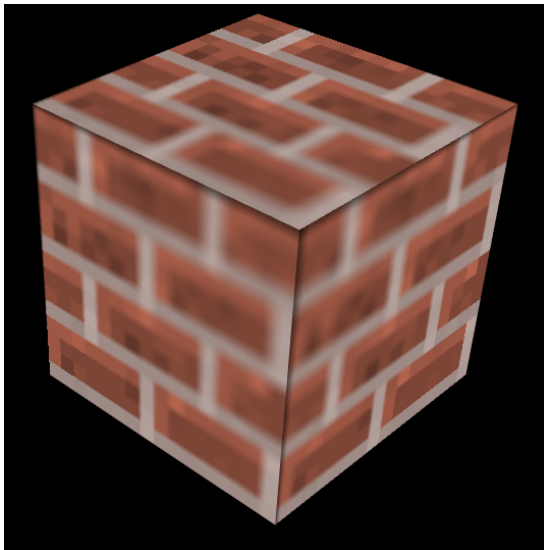
Portion of code of fragment shader (Scatter as Gather)

```
uniform sampler2D u_Sampler; uniform vec2 u_TexResolution;
uniform float u_d, u_vf; uniform int u_MaxBlur;
in vec2 v_TexCoord; in vec4 v_Position; out vec4 fragColor;
void main() {
    float vo = length(vec3(v_Position));
    int c = min(int(u_d*abs(vo/u_vf - 1.0)), u_MaxBlur);
    vec3 color = vec3(0.0); int total = 0;
    for (int i= -c; i <= c; ++i)
        for (int j= -c; j <= c; ++j)
            if (i*i + j*j <= c*c) {
                color += texture(u_Sampler, v_TexCoord
                                + vec2(i, j) / u_TexResolution).rgb;
                total++;
            }
    fragColor = vec4(color / float(total), 1.0);
}
```

Artifacts of the initial stochastic approach



WebGL scene using scatter-as-gather approach

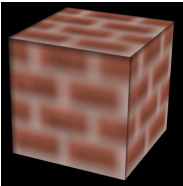
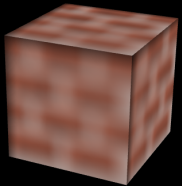

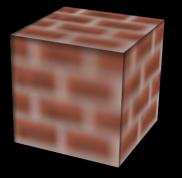


Further variations

We implemented different formulas for the circle of confusion calculation with small variations between them.

	linear	hyperbolic
absolute	$d \cdot \left \frac{v_o}{v_f} - 1 \right $	$d \cdot \left \frac{v_f}{v_o} - 1 \right $
squared	$d \cdot \left(\frac{v_o}{v_f} - 1 \right)^2$	$d \cdot \left(\frac{v_f}{v_o} - 1 \right)^2$

A visualization of the different variations

	linear	hyperbolic
absolute		
squared		

Three.js

The aim of three.js is to create lightweight, cross-browser, general purpose 3D library.

Three.js is often confused with WebGL since three.js uses WebGL to draw 3D. WebGL is a very low-level system that only draws points, lines, and triangles.

Three.js implementation

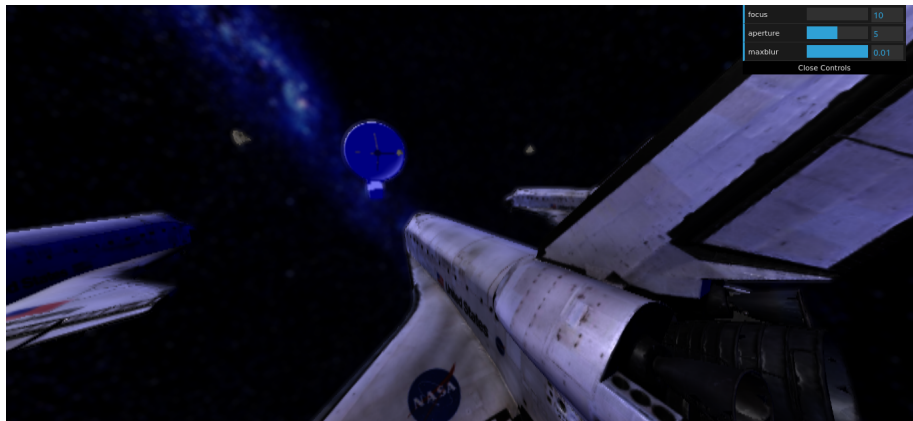
The three.js implementation uses the linear formula using the absolute value for the circle of confusion:

$$c = d \cdot \left| \frac{v_o}{v_f} - 1 \right|$$

It also uses the scatter-as-gather approach, meaning that the blur changes continuously.

The average color is calculated using gaussian weights, as this makes the calculations more time efficient.

Three.js scene



GitHub page

To see the implementation in action, visit:

https://marcobenelli.github.io/DepthOfField_WebGL/

In there you will also find the link to the source code.

WebGL version

The **WebGL version** uses only tools available in pure WebGL. It is therefore a low level implementation using OpenGL shaders written in GLSL.

In this version we can choose either the **initial stochastic approach** or the **scatter-as-gather approach**, as described in [Practical Post-Process Depth of Field](#) article by NVIDIA.

The lens parameters, that the users can vary, are the **focus length**, the **focal aperture** and the **maximum blur**. We can also independently set several other parameters including **hyperbolic blur** and **squared blur**.

Three.js version

The **Three.js version** uses the **three.js** library. This is a high level implementation which means that we are able to easily add glTF models.

The models are taken from **Sketchfab**, the leading platform for 3D & AR on the web.

This version uses only the **scatter-as-gather** approach with a gaussian blur.

The lens parameters, that the users can vary, are the **focus length**, the **focal aperture** and the **maximum blur**.

All of the code is available on [GitHub](#)

Conclusion

Having implemented all of these different approaches and variations, we can say that the scatter-as-gather approach is really beneficial to the realism of the scene, while other parameters are less important. Three.js is a powerful tool when dealing with models, but to fully understand it, you should also be familiar with pure WebGL.