Three.js implementation of Depth Of Field and Motion Blur

Computer Graphics and 3D Course Project

Lorenzo Agnolucci, Federico Nocentini

Supervisor: Prof. Stefano Berretti

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze

Index

Introduction

Introduction

- In the physical world any real camera has a finite aperture and shutter speed, as well as a lens to focus the incoming light
- A pinhole camera used in CG does not need these features, but they can be simulated to mimic the behaviour of a real camera
- We implemented both Depth of Field and Motion Blur as post-processing effects to increase the realism of the scene
- In particular our DoF implementation physically accurately models the typical DoF effect of a real camera

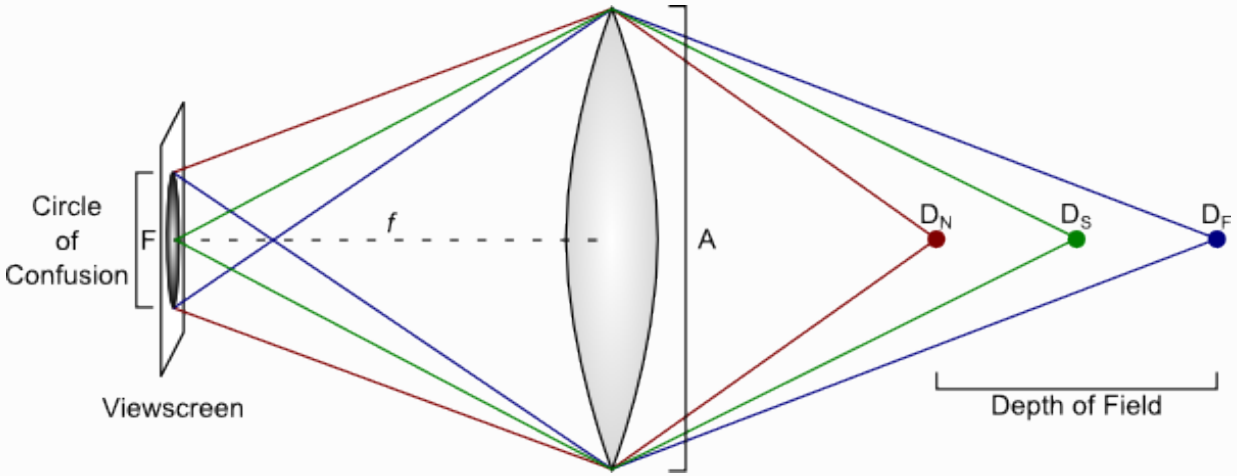
Introduction
Technologies used

- Our implementation is built with WebGL and three.js
- Three.js is a cross-browser JavaScript library and API based on WebGL and used to create and display animated 3D computer graphics in a web browser
- EffectComposer is a class included in three.js that conveniently manages a chain of post-processing passes, abstracting over framebuffer objects

Depth of Field

Description

- Depth of field is the effect in which objects within some range of distances in a scene appear in focus, and objects nearer or farther than this range appear out of focus
- A light source at a certain distance (known as plane in focus) from the camera lens converge to a single point on the film
- Anything that is not at this distance projects to a region called circle of confusion (CoC)



CoC formula

- The CoC is computed with the formula:

$$CoC = \left| A \cdot \frac{(F \cdot (D - P)}{(D \cdot (P - F)} \right|$$

- where:
  - A is the aperture
  - F is the focal length
  - D is the object distance
  - P is the plane in focus
- Alternatively we could substitute the aperture with the f-stop N, with $A = \frac{F}{N}$

Outline

1. Render the scene to a **render target** to get the original and the depth textures

2. Compute the **Circle of Confusion** using the camera parameters and the linearized depth of the vertex

3. **Blur** the CoC of **foreground** objects to avoid sharp silhouettes

4. Blur the rendered texture and **blend** it with the original one according to the CoC

Basic vertex shader

- This basic vertex shader is shared and used by almost all of the following fragment shaders

- It does not modify the scene geometry, it simply computes the vertex position using the model, view and projection matrices

```
1 out vec2 vUv;
2
3 void main() {
4     vUv = uv;
5     gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
6 }
```

Depth visualization and linearization

Fragment shader


```glsl
#include <packing>

uniform sampler2D tDepth;   // Depth texture
uniform float zNear;    // Camera near plane
uniform float zFar;     // Camera far plane

in vec2 vUv;

float readDepth(sampler2D depthSampler, vec2 coord) {
    float fragCoordZ = texture2D(depthSampler, coord).x;
    // perspectiveDepthToViewZ -> (zNear * zFar) / ((zFar - zNear) * fragCoordZ - zFar)
    float viewZ = perspectiveDepthToViewZ(fragCoordZ, zNear, zFar);
    // viewZToOrthographicDepth -> (viewZ + zNear) / (zNear - zFar)
    return viewZToOrthographicDepth(viewZ, zNear, zFar);
}

void main() {
    float depth = readDepth( tDepth, vUv );
    gl_FragColor.rgb = vec3( depth );
}
```

CoC

Fragment shader


```
uniform sampler2D tDepth;   // Depth texture
uniform float zNear;    // Camera near plane
uniform float zFar;     // Camera far plane
uniform float focalDepth;       // Focal distance value in meters
uniform float focalLength;      // Focal length in mm
uniform float fstop;        // F-stop value
uniform bool mouseFocus;
uniform vec2 mouseCoords;
in vec2 vUv;

void main() {
    float depth = readDepth( tDepth, vUv );
    float focalDepthMM = focalDepth * 1000.0;       // Focal depth in millimeters
    if(mouseFocus){
        float mouseDepth = readDepth(tDepth, mouseCoords);
        float mouseFocalDepth = - zFar * zNear / (mouseDepth * (zFar - zNear) - zFar);
        focalDepthMM = mouseFocalDepth * 1000.0;
    }
    float objectDistance = - zFar * zNear / (depth * (zFar - zNear) - zFar);
    objectDistance = objectDistance * 1000.0;       // Object distance in millimeters
    float CoC = (focalLength / fstop) * (focalLength * (objectDistance - focalDepthMM))
                / (objectDistance * (focalDepthMM - focalLength));
    CoC = clamp(CoC, -1.0, 1.0);
    if(CoC >= 0.0){
        gl_FragColor.b = CoC;
    } else{
        gl_FragColor.g = abs(CoC);
}}
```

Blur foreground

Fragment shader

```glsl
uniform sampler2D tDiffuse;     // CoC texture
uniform float widthTexel;       // 1.0 / window.innerWidth
uniform float heightTexel;      // 1.0 / window.innerHeight
uniform bool horizontalBlur;        //True -> Horizontal, False -> Vertical
uniform float weight[5];

in vec2 vUv;

void main() {
    vec2 blurDirection;
    if (horizontalBlur){
        blurDirection = vec2(widthTexel, 0.0);
    }
    else {
        blurDirection = vec2(0.0, heightTexel);
    }

    float fragColorGreen = texture2D(tDiffuse, vUv).g * weight[0];

    for (int i = 1; i < 5; i++){
        fragColorGreen += texture2D(tDiffuse, (vUv+float(i)*blurDirection)).g*weight[i];
        fragColorGreen += texture2D(tDiffuse, (vUv-float(i)*blurDirection)).g*weight[i];
    }

    gl_FragColor.g = fragColorGreen;
    gl_FragColor.b = texture2D(tDiffuse, vUv).b;

}
```

DoF

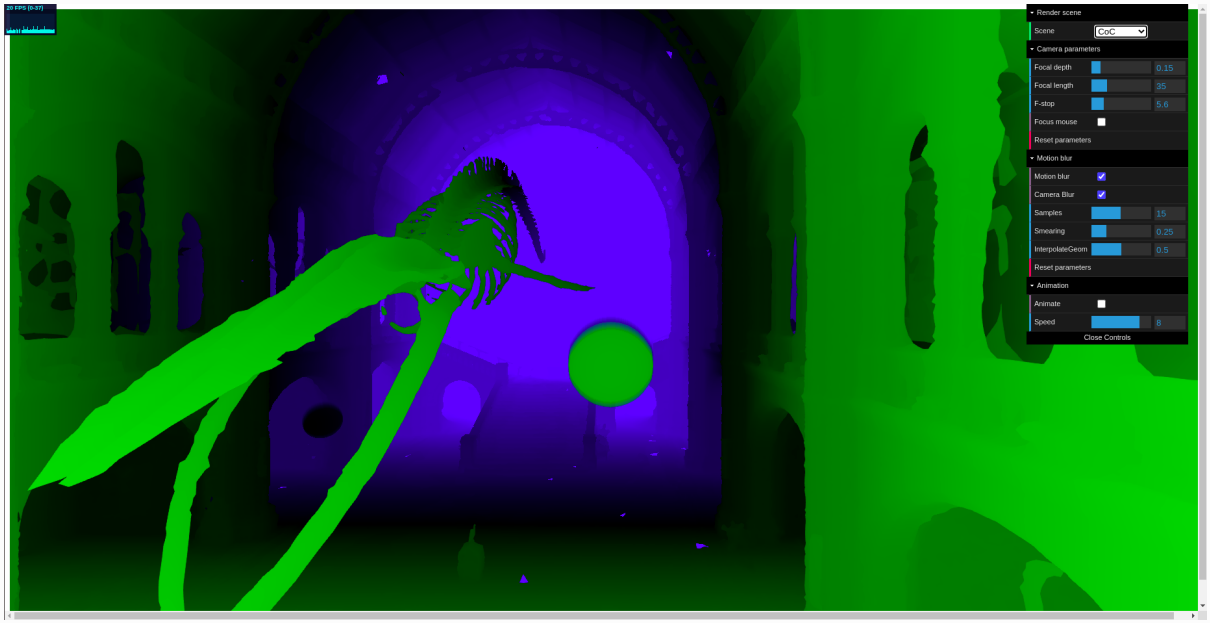
Fragment shader

```
uniform sampler2D tDiffuse;     // CoC texture
uniform sampler2D tOriginal;    // Basic rendering texture
uniform float bokehBlurSize;
uniform float widthTexel;
uniform float heightTexel;
uniform bool dofEnabled;
uniform bool showFocus;

in vec2 vUv;

float getWeight(float dist, float maxDist){
    return 1.0 - dist/maxDist;
}

void main() {
    vec3 sourceColor = texture2D(tOriginal, vUv).rgb;
    float minCoC = 0.1;
    gl_FragColor.rgb = sourceColor;
    float CoC = max(texture2D(tDiffuse, vUv).g, texture2D(tDiffuse, vUv).b);
    if(dofEnabled){
        if (CoC > minCoC){
            float bokehBlurWeightTotal = 0.0;
            vec3 blurColor = vec3(0.0);
            for (float i=-bokehBlurSize; i<bokehBlurSize+1.0; i++){
                for (float j=-bokehBlurSize; j<bokehBlurSize+1.0; j++){
                    vec2 dir = vec2(i, j) * vec2(widthTexel, heightTexel);
                    float dist = length(dir);
                    if (dist > bokehBlurSize){ continue; }
```

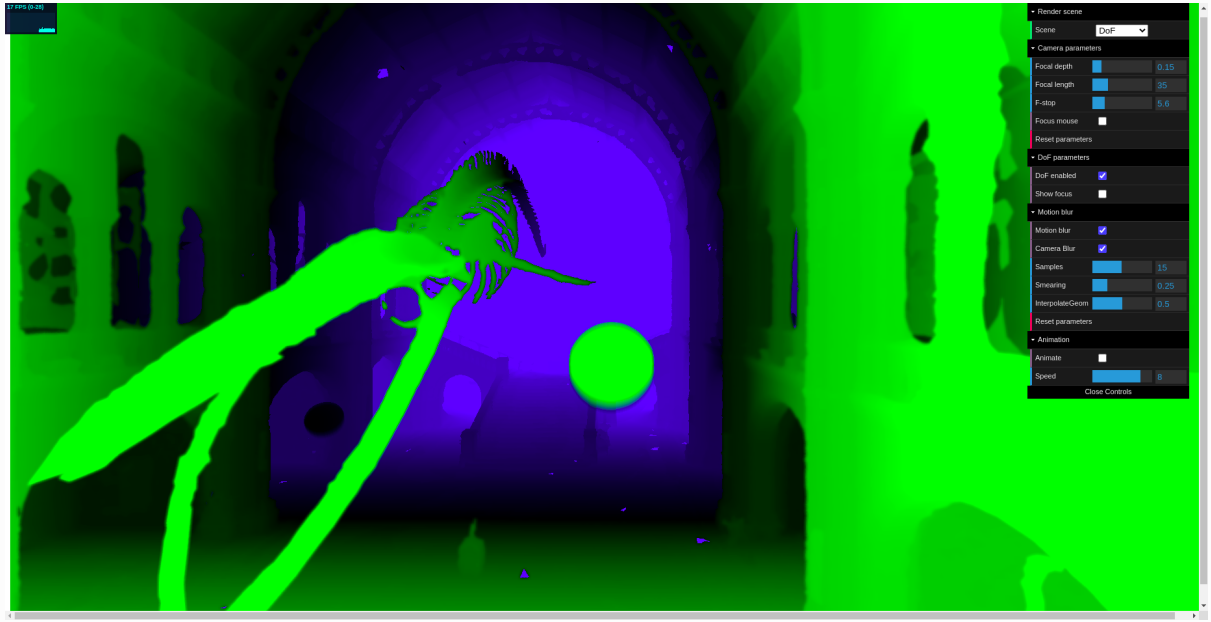
DoF
Fragment shader (cont.)


```
29                      vec2 curUv = dir + vUv;
30                      float curCoC = max(texture2D(tDiffuse, curUv).g,
31                                         texture2D(tDiffuse, curUv).b);
32                      if(curCoC > minCoC){
33                          float weight = getWeight(dist, bokehBlurSize);
34                          bokehBlurWeightTotal += weight;
35                          blurColor +=  weight * texture2D(tOriginal, curUv).rgb;
36                      }
37                  }
38              }
39              blurColor /= bokehBlurWeightTotal;
40
41              gl_FragColor.rgb = mix(sourceColor, blurColor, CoC);
42          }
43          else{
44              if(showFocus){
45                  gl_FragColor.rgb = mix(sourceColor, vec3(0.988, 0.596, 0.011),
46                                         (1.0 - CoC * 10.0));
47              }
48          }
49      }
50  }
```

Example images



Example images



FLORENTINA STUDIORUM UNIVERSITAS

Motion Blur

Description

- Motion blur represents the blurring of moving objects in the direction of motion
- It is caused by the fact that in real cameras the film is exposed to a moving scene while the shutter is open
- It is often uses in games because it gives the sensation of speed and also helps smooth out a game's appearance

Outline

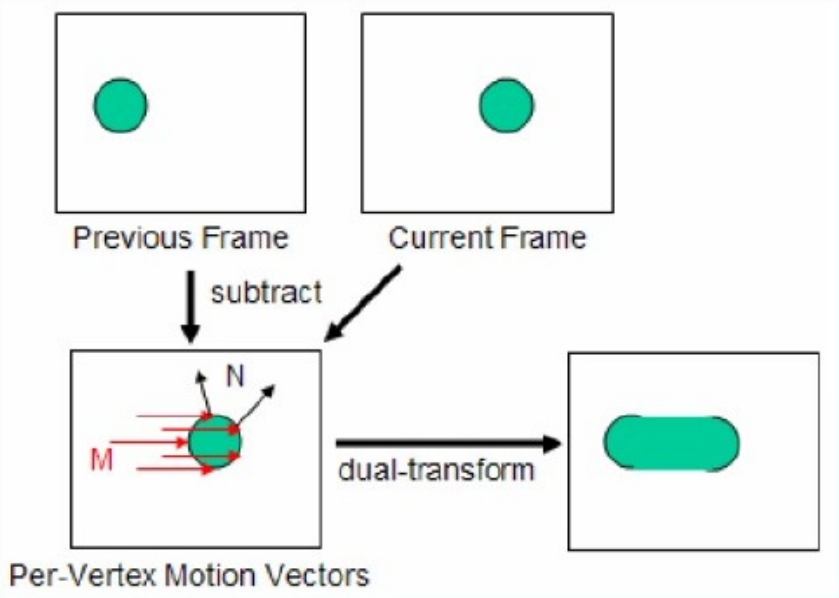
1. Render the scene to a **render target** to get the original texture (in our case DoF is already present)

2. **Compute** the **velocity** at each pixel, that is the difference between the current and the previous position

3. **Sample** the original texture based on the motion vector and compute a **weighted sum** to obtain the final pixel color

Computing velocity

- Transform each vertex with current and previous matrices and compute the difference between the positions
- To avoid a null velocity outside the object silhouette we use Matthias Wloka's trick to stretch object geometry, that is:
  - Compare normal direction with motion vector using dot product
  - If normal is pointing in direction of motion, transform the vertex by the current transform, otherwise use previous transform



Velocity
Vertex shader

```
uniform mat4 prevProjectionMatrix;
uniform mat4 prevModelViewMatrix;
uniform float interpolateGeometry;

out vec4 prevPosition;
out vec4 newPosition;

void main() {
    // Get the normal
    vec3 transformedNormal = normalMatrix * vec3(normal);

    newPosition = modelViewMatrix * vec4(position, 1.0);
    prevPosition = prevModelViewMatrix * vec4(position, 1.0);

    // Delta between frames
    vec3 delta = newPosition.xyz - prevPosition.xyz;
    vec3 direction = normalize(delta);

    // Stretch along the velocity axes
    float stretchDot = dot(direction, transformedNormal);

    newPosition =  projectionMatrix * newPosition;
    prevPosition = prevProjectionMatrix * prevPosition;
    gl_Position = mix(newPosition, prevPosition, interpolateGeometry *
                    (1.0 - step(0.0, stretchDot)));
}
```

Velocity
Fragment shader


```glsl
uniform float smearIntensity;

in vec4 prevPosition;
in vec4 newPosition;

void main() {
    vec3 vel;
    vel = (newPosition.xyz / newPosition.w) - (prevPosition.xyz / prevPosition.w);
    gl_FragColor = vec4(vel * smearIntensity, 1.0);
}
```

Geometry dilation visualization

Vertex shader

```
uniform mat4 prevProjectionMatrix;
uniform mat4 prevModelViewMatrix;
uniform float interpolateGeometry;

out vec3 color;

void main() {
    --- Same code of Velocity vertex shader ---

    color = (modelViewMatrix * vec4(normal.xyz, 0)).xyz;
    color = normalize(color);
}
```

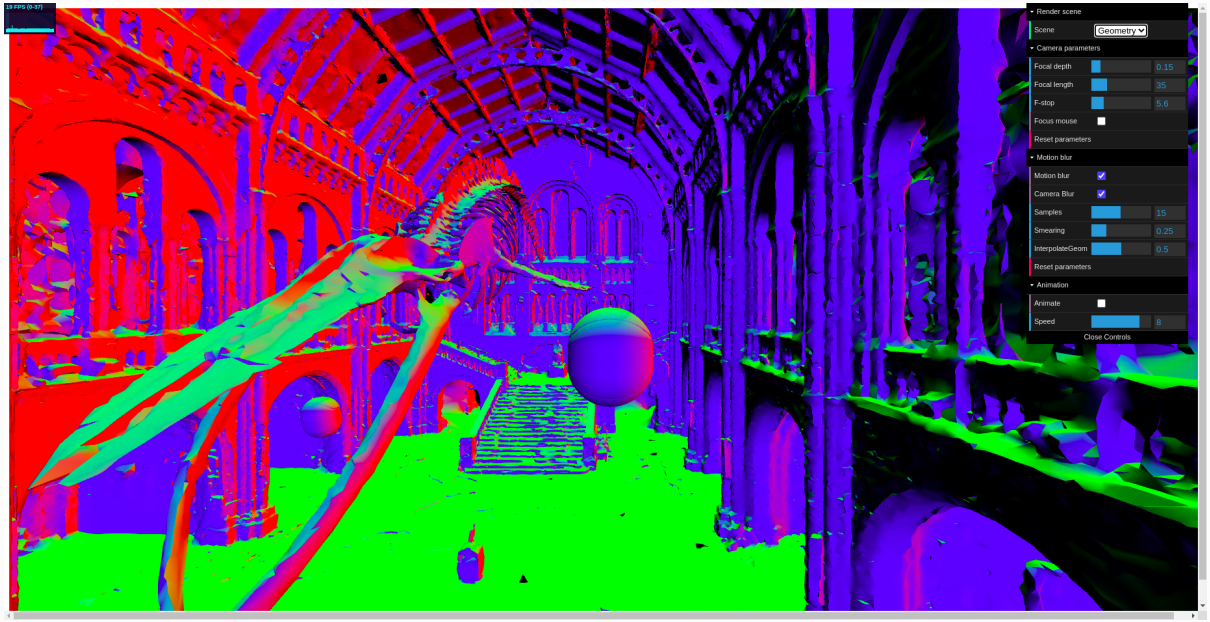
Fragment shader

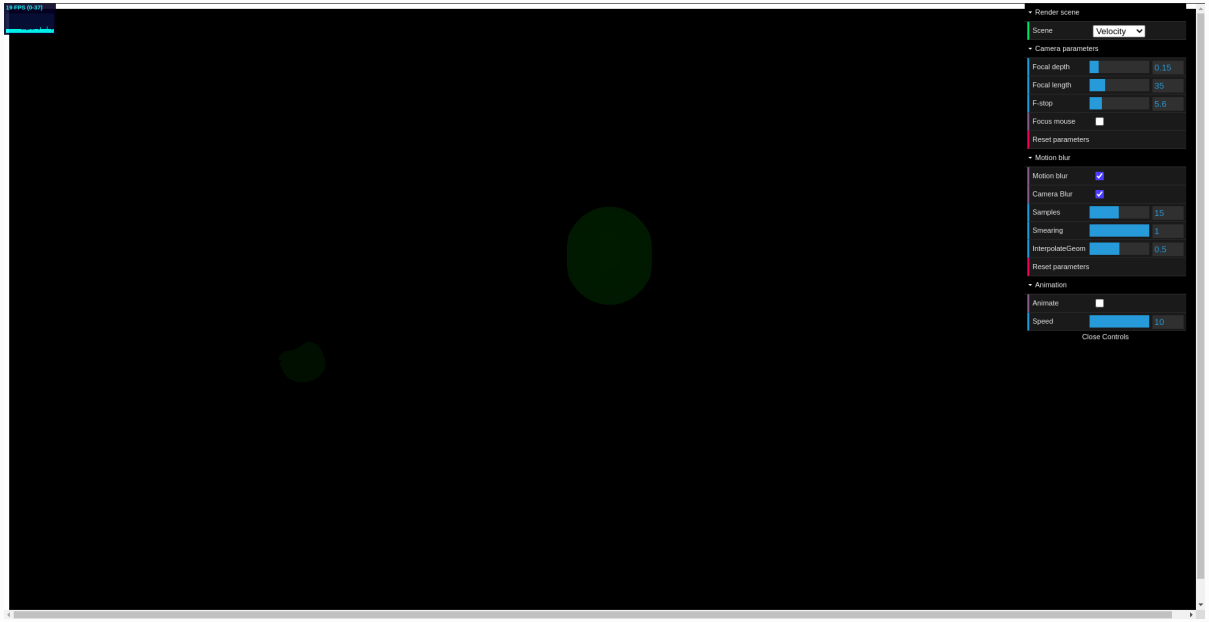
```
in vec3 color;

void main() {
    gl_FragColor = vec4(color, 1);
}
```

Motion Blur
Fragment shader


```glsl
uniform sampler2D sourceBuffer;     // DoF texture
uniform sampler2D velocityBuffer;   // Velocity texture

in vec2 vUv;

void main() {
    vec2 vel = texture2D(velocityBuffer, vUv).xy;
    vec4 result = texture2D(sourceBuffer, vUv);
    for(int i = 1; i <= SAMPLES; i++) {
        vec2 offset = vel * (float(i - 1) / float(SAMPLES) - 0.5);
        result += texture2D(sourceBuffer, vUv + offset);
    }
    result /= float(SAMPLES + 1);
    gl_FragColor = result;
}
```

Example images





Example images

Effect Composer

Add all the passes to the EffectComposer:

```
1  const DoFComposer = new EffectComposer(renderer);
2  const CoCPass = new ShaderPass(CoCShader);
3  DoFComposer.addPass(CoCPass);
4  const verticalBlurPass = new ShaderPass(verticalBlurShader);
5  DoFComposer.addPass(verticalBlurPass);
6  const horizontalBlurPass = new ShaderPass(horizontalBlurShader);
7  DoFComposer.addPass(horizontalBlurPass);
8  const DoFPass = new ShaderPass(DoFShader);
9  DoFComposer.addPass(DoFPass);
10 const motionPass = new MotionBlurPass(scene, camera, motionBlurParameters);
11 DoFComposer.addPass(motionPass);
12 const antialiasingPass = new ShaderPass(FXAAShader);
13 antialiasingPass.renderToScreen = true;
14 DoFComposer.addPass(antialiasingPass);
```

And in the render loop:

```
1  renderer.setRenderTarget(basicTarget);
2  renderer.render(scene, camera);
3  CoCPass.uniforms.tDiffuse.value = basicTarget.texture;
4  CoCPass.uniforms.tDepth.value = basicTarget.depthTexture;
5  DoFPass.uniforms.tOriginal.value = basicTarget.texture;
6  DoFComposer.render();
```

Conclusions

Conclusions and future work

- We implemented Depth of Field and Motion Blur as post-processing effects to mimic the behaviour of a real camera and increase the realism of the scene
- Our DoF implementation physically accurately models the typical DoF effect of a real camera
- As a future work it would be interesting to implement a more sophisticated motion blur [1]

[1] A reconstruction filter for plausible motion blur, *https://casual-effects.com/research/McGuire2012Blur/McGuire12Blur.pdf*