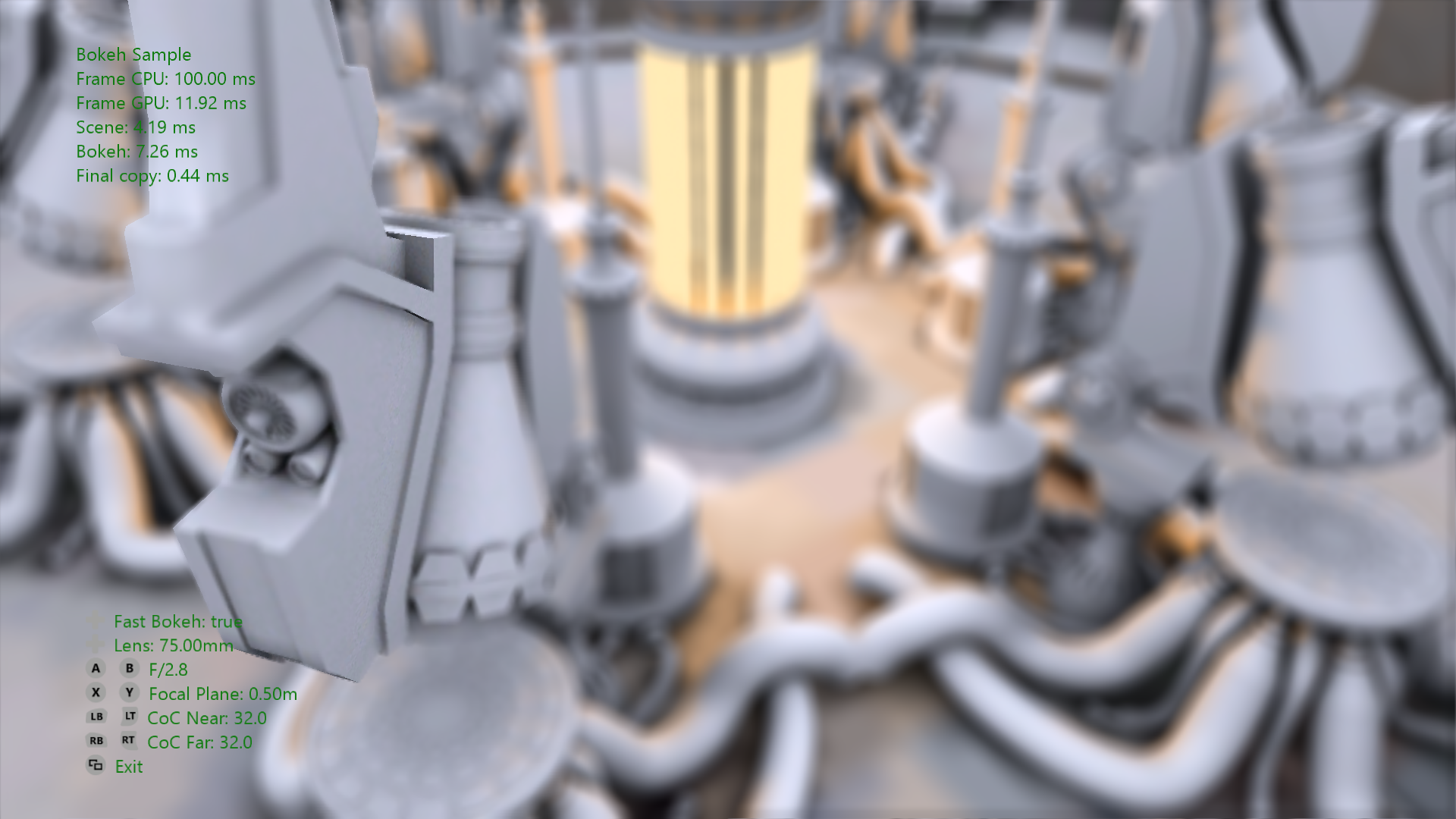


Bokeh Sample (DirectX 11)

*\* This sample is compatible with the August 2016 Xbox One XDK.*

# Description

This sample demonstrates how to create a depth of field effect using point sprites rendering.



# Using the sample

The sample uses the following controls.

## Controls

|  |  |
| --- | --- |
| Action | Gamepad |
| Camera Rotation | Left Thumbstick X-axis |
| Camera Elevation | Left Thumbstick Y-axis |
| Camera Distance | Right Thumbstick Y-axis |
| Focal Length | D-Pad Left/Right |
| Focal Plane | X Button / Y Button |
| F-Stop | A Button / B Button |

|  |  |
| --- | --- |
| Action | Gamepad |
| Maximum Near CoC Size | Left Shoulder / Left Trigger |
| Maximum Far CoC Size | Right Shoulder / Right Trigger |
| Toggle Fast Bokeh Shader | D-Pad Up |
| Cycle Preset Shots | D-Pad Down |
| Exit Sample | View Button |

# Implementation notes

Blurring of the out-of-focus areas in a picture taken by a camera is an important effect that draws viewer's attention to a certain area of the image. The area of apparent sharpness in the image is called depth of field (DOF) and the areas in front of it and at the back appear blurred.

It is not surprising that such an important artistic tool has found its way in computer generated graphic images. In photography, DOF is determined by lens focal length, aperture and the distance to the subject and is approximately given by a thin lens equation. In computer graphics it is possible to use arbitrary parameters and formulae for DOF, although it is convenient to define them as in photography.

DOF exists because a point in the world is projected into a circle onto a film or a sensor of the camera when viewed though a lens and the rest of the imaging system. It is always a circle, albeit a tiny one in sharp in-focus areas or clipped in the areas with very large blurs. That circle is called Circle of Confusion (CoC).

Commonly, real time graphics implementations of DOF, calculate the CoC per pixel, and then blur the image accordingly (offline DOF implementations work in a different way). Such real-time blur needs to satisfy the following restrictions to look realistic and be useful.

1. Each point is projected on the screen as an image of the camera's iris, that has a size of CoC. The resulting look is commonly referred to as Bokeh DOF.

2. Blurred points should be blended back to front so that at the very least the in-focus image is never obscured by the blurred circles caused by the farther-from-camera pixels and the blurred circles caused by the nearer-to-the-camera pixels blend on top of the in-focus and far out-of-focus images. Absence of this feature causes colour bleeding around the edges of in-focus part of the image.

3. Partial occlusion – large apertures allow the optical systems to see around obstacles. This means that larger CoC in the near field will reveal objects underneath.

4. It should be possible to have very large sizes of CoC and it should be fast.

The algorithm demonstrated in the sample satisfies all points listed above except number 3. To correctly handle partial occlusion, we need to either render the scene behind the occluders (similar to depth peeling) or smear the color from the occluded object into the partial occlusion area. This sample doesn’t perform that step for performance reasons, instead taking the underlying pinhole image, which may cause slight visual artifacts in some places.

## Algorithm:

The concept is simple — we take a pixel of the input, calculate the CoC of it, and output a point sprite that is the size of CoC and colored as the source pixel. We sort them in the order of occlusion and accumulate all such sprites and get the correctly blurred image.

If implemented in such a straightforward way, the effect would look fantastic, but it can't be made real time on the modern hardware, so below is the optimized algorithm.

1. Convert source color and depth to RGBZ texture. The source textures are of the size W\*2 x H\*2

2. Downsample that texture once, taking average of 4 colors and the minimum value of 4 depths.

3. For a WxH downsampled texture, set up a render target with 6 viewports — a pair of each: WxH, W/2xH/2, and W/4xH/4

4. Render W\*H/4-point primitives

1. For each point primitive read 4 RGBZ values of the downsampled source texture

2. If the 4 pixels don't differ much, output one sprite, otherwise output 4 sprites

3. Depending on the size of the sprite route it into a corresponding viewport, sprites nearer than in-focus area going into the "near" set of viewports, the ones farther go into the "far" set viewport

5. After finishing rendering the 6 viewports, recombine them with the in-focus image, making sure in-focus pixels override the far blurred area and the near pixels blend over the in-focus and far pixels

To improve blending of the near blurred pixels, image energy conservation is important. Each source pixel has unit energy, so after splatting it with a point sprite, that energy is distributed over the area of the sprite. Due to different iris textures and rasterization rules it's impossible to calculate the weight in closed form for all sizes and sprite origins. The sample renders the iris sprite at different sizes and calculates the resulting weight, then computes the normalization factor.

# Update history

2015: Original sample authoring

November 2018: rewritten for new sample template

# Privacy Statement

When compiling and running a sample, the file name of the sample executable will be sent to Microsoft to help track sample usage. To opt-out of this data collection, you can remove the block of code in Main.cpp labeled “Sample Usage Telemetry”.

For more information about Microsoft’s privacy policies in general, see the [Microsoft Privacy Statement](https://privacy.microsoft.com/en-us/privacystatement/).