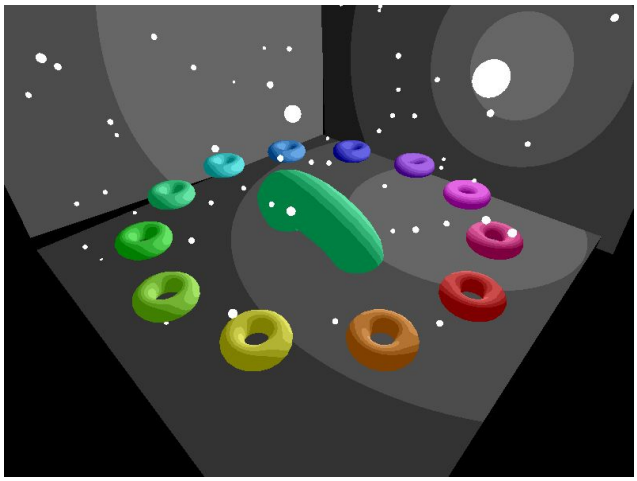
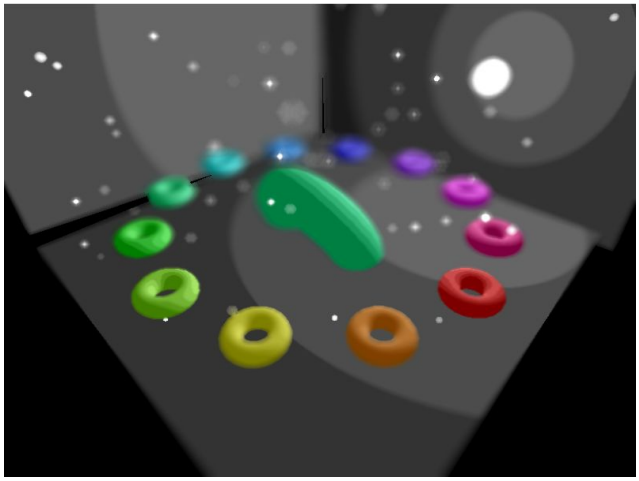
Lab 3: Depth of Field

Objective

Create the necessary shader programs to implement the Depth of Field (DoF) effect. The images on the right show: a scene rendered without DoF enabled (top) and with DoF enabled (bottom).



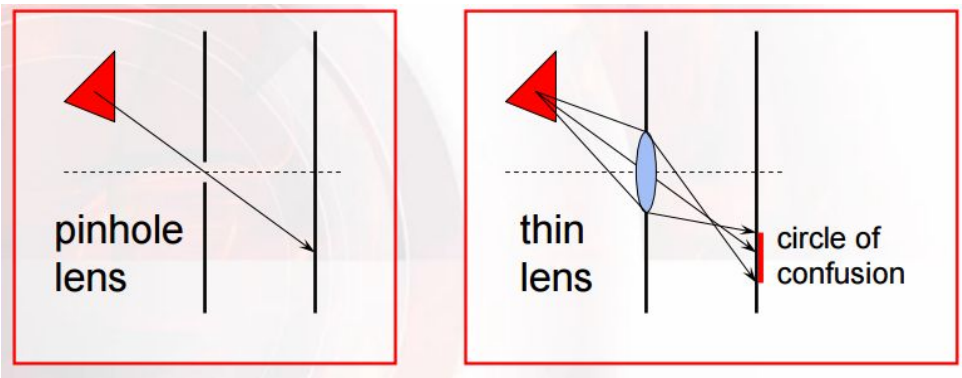


Background

We will be implementing the Depth of Field algorithm described in the “Efficiently Simulating the Bokeh of Polygonal Apertures in a Post-Process Depth of Field Shader” paper. Doom (2016) uses a modified version of this algorithm! The various definitions and descriptions used throughout this document derived from that paper.

Games are rendered using a “pinhole” camera model. This means that for every pixel in the rendered image, there is one and only one corresponding point in the scene. This means that every pixel in the rendered images are in perfect focus. Such cameras do not exist in reality, in the real world an image is formed from light rays passing through a camera lens and hitting a imaging sensor or film, there can be several light rays which contribute to a pixel’s color. The image below demonstrates this. Whether or not a pixel is in focus depends on the size of the pixel’s **circle of confusion**.

The basic idea behind the DoF algorithm is that we will be taking our perfectly in focus image as rendered by our pinhole camera and adding the circle of confusion imperfections to it.



Left: The traditional pinhole lens, a pixel's color is determined from a single light ray.
Right: Camera with a thin lens, a pixel's color is determined from several rays

Starting Point

The C++ side of the framework is mostly setup for you. You will need to create the necessary framebuffer objects and send the appropriate uniforms to the shaders.

Ensure that your project toggles the following modes appropriately:

- '1' = Default shading (already done!)
- '2' = Horizontal Bokeh Pass
- '3' = Diagonal Bokeh Pass A
- '4' = Diagonal Bokeh Pass B
- '5' = Composited Depth of Field

Part 0: Initialization

The starter code has no frame buffer objects created. You will need four FBOs to implement depth of field:

1) "Scene fbo" this is where the scene geometry will be rendered to. Make sure this has a depth buffer.
2) "Horizontal Bokeh fbo" (See pass 2)
3) "Diagonal Bokeh fbo A" (See pass 3)
4) "Diagonal Bokeh fbo B" (See pass 4)

These FBOs only need a single color attachment.

Note that we are using floating point FrameBufferObjects meaning the internal format used when creating the FBO textures is GL_RGBA32F which means each channel is stored as a 32 bit float. This is to ensure that the blurriness value calculated in pass 1 does not get truncated.

Initialize your FBOs in the initializeFrameBuffers function found in main.cpp

Part 1: The Scene Pass

Up until now our scene pass simply rendered our geometry, performed lighting and stored the computed pixel colours in an FBO. In this DoF implementation, we will be calculating the pixel's blurriness in addition to the pixel's color. The RGB channels of our framebuffer will contain the pixels color and the alpha channel (A) will contain the pixels blurriness. The pixel's blurriness can be calculated using the formula below:

$$c = A \cdot \frac{|S_2 - S_1|}{S_2} \cdot \frac{f}{S_1 - f}$$

c is the diameter of the circle of confusion
A is the diameter of the aperture
S2 is the pixel's depth
S1 is the distance to the focal plane (the plane in the scene which is in focus, so if this set to 10.0 then this means the pixels which are exactly 10 units away from the camera will be in focus)
f is the camera's focal length

Here is the code to calculate the pixel's blurriness:

```
//--------------------------------------------------------------------------
// CoCMap_PS
// Lorne McIntosh, 2011
//--------------------------------------------------------------------------
//      Computes the Circle of Confusion diameter at every pixel. This is
//      then converted into a % of the total image size, by assuming an image
//      sensor (or film) 24mm in height (35mm film standard). To avoid artifacts,
//      this % is then artificially clamped to MaxCoC. The result should be stored
//      in the alpha channel of the color map to be read by subsequent passes.
//--------------------------------------------------------------------------
half4 CoCMap_PS(QuadVertexOutput IN,
        uniform sampler2D DepthSamp,
        uniform half A,         //aperture
        uniform half f,         //focal length
        uniform half S1,        //focal distance
        uniform half Far,       //far clipping plane
        uniform half MaxCoC     //max CoC diameter
) : COLOR
{
        //reconstruct scene depth at this pixel
        const half S2 = tex2D(DepthSamp, IN.UV).x * Far;

        //calculate circle of confusion diameter
        //(from http://en.wikipedia.org/wiki/Circle_of_confusion)
        const half c = A * (abs(S2 - S1) / S2) * (f / (S1 - f));

        //define height of camera's image sensor (width is assumed from
        //aspect ratio). (35mm "full-frame" film format is 36mm x 24mm)
        const half sensorHeight = 0.024f;       //24mm

        //put CoC into a % of the image sensor height
        const half percentOfSensor = c / sensorHeight;

        //artificially clamp % between 0 and MaxCoC
        const half blurFactor = clamp(percentOfSensor, 0.0f, MaxCoC);

        return blurFactor;
}
```

This code is in HLSL which is a different shading language. You will not be able to just take this code and use it as is, make the appropriate changes to make it work.

Here are the parameters I used to take all of the images in this document:

A = 0.6062
f = 0.0303
S1 = 20.0

Use those values as you implement the algorithm, once you have everything working try playing around with these numbers and observe how depth of field effect changes. You can either hard-code these values in the shader, or you can send them as uniforms.

Part 2: Horizontal Bokeh Pass

Now that we have our scene's color and blurriness stored in a texture, we can begin the process of implementing the Bokeh Depth of Field. For the bokeh_f.glsl shader you will need to set the following uniforms to the appropriate values:

uniform sampler2D u_depth;

- This is the scene's depth texture, use the FBO::bindDepthTextureForSampling function to bind FBO's depth texture

uniform sampler2D u_tex;

- This is the scene's color (RGB) + blurriness (A) texture

uniform vec4 u_cameraParams;

- These are some parameters needed by the algorithm:
    - u_cameraParams.y = the camera's aspect ratio (window width / window height)
        - Remember to cast the window width and height to a float when you perform the division!
    - u_cameraParams.x = angle of the blur, for the horizontal pass this will be set to zero

Once you have the uniforms all set up, you can begin writing the actual shader.

The first step is to figure out which pixels we want to sample. This filter is essentially a directional blur. When we implemented the Box Blur filter, we sampled from each of the neighbouring pixels which gave us a uniform blur. With this filter we only want to sample from the pixels in the direction we are blurring in. So when we perform the horizontal blur, we only sample from the pixels to the left and right of the current pixel being processed.

Here is the code to calculate the offsets:

```
//--------------------------------------------------------------------------
// makeOffsets
// Lorne McIntosh, 2011
//--------------------------------------------------------------------------
//      Creates linear sample offsets given an angle in radians
//      This would normally be done in the application once & stored. The values
//      would then be passed to the shader via constant registers.
//--------------------------------------------------------------------------
OffsetData makeOffsets(half angle)
{
        OffsetData output;

        const half aspectRatio = ViewportSize.x / ViewportSize.y;

        half radius = 0.5f;

        //convert from polar to cartesian
        half2 pt = half2(radius * cos(angle), radius * sin(angle));

        //account for aspect ratio (to avoid stretching highlights)
        pt.x /= aspectRatio;

        //create the interpolations
        for(int i = 0; i < numSamples; i++)
        {
                half t = i / (numSamples - 1.0f); //0 to 1
                output.offsets[i] = lerp(-pt, pt, t);
        }

        return output;
}
```

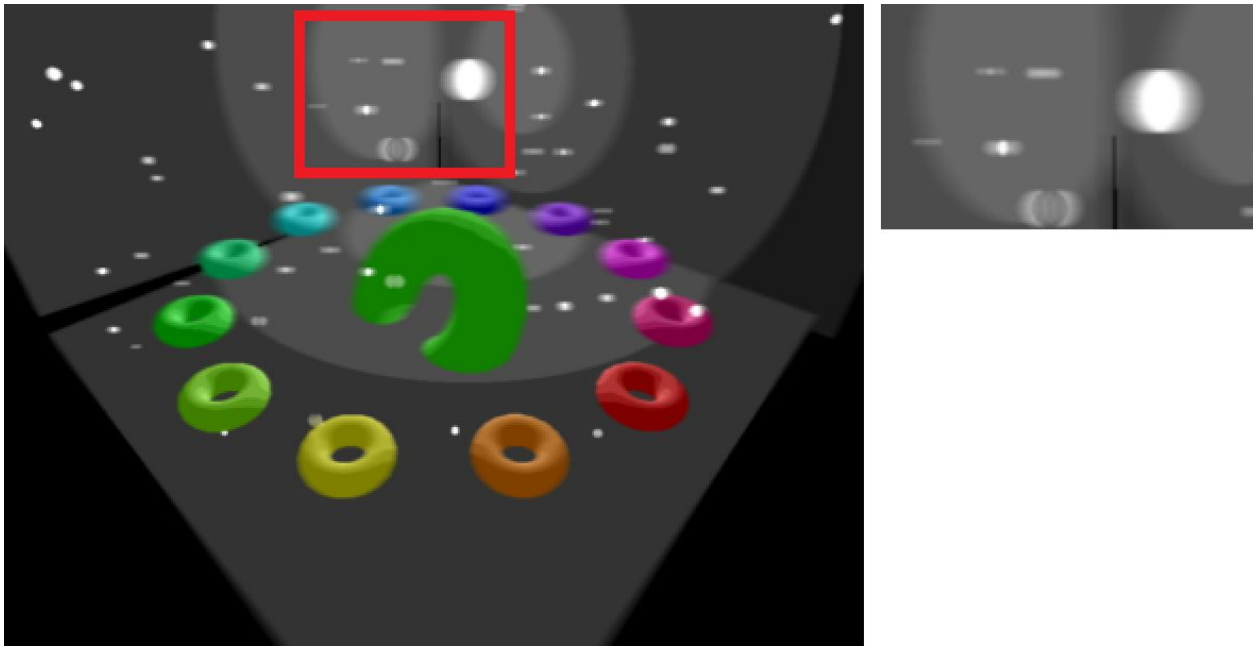
Note that in a proper implementation you would calculate these offset once CPU side and send the offsets up to the shader as an array of uniforms, but for the sake of making things a little easier on ourselves we will be creating them in the fragment shader (which is the worst place to do it!).

The angle parameter allows us to specify the angle of the blur. In the horizontal pass we will be using an angle of zero. Note: make sure the angle you pass to the shader is in radians!

Once you have implemented the makeOffsets function, you need to perform the actual sampling and blurring **on the scene texture.** The authors of this algorithm were kind enough to give us the code for this too:

```
//--------------------------------------------------------------------------
// DepthOfField_PS
// Lorne McIntosh, 2011
//--------------------------------------------------------------------------
//      A bokeh-producing depth of field pixel shader. This performs one pass of a
//      seperable filter and must be used twice (with different offsets) to create
//      a cumulative effect. The alpha channel of ColorSamp must hold a CoC
//      (i.e. bluriness) map. See the paper for more info.
//--------------------------------------------------------------------------
half4 DepthOfField_PS(QuadVertexOutput IN,
        uniform sampler2D ColorSamp,
        uniform sampler2D DepthSamp,
        uniform OffsetData offsetData
) : COLOR
{
        //these are used to tune the "pixel-bleeding" fix
        const half bleedingBias = 0.02f;
        const half bleedingMult = 30.0f;

        //get the center samples for later reference
        half4 centerPixel = tex2D(ColorSamp, IN.UV);
        half centerDepth = tex2D(DepthSamp, IN.UV);

        //for finding the weighted average
        half4 color = 0.0f;
        half totalWeight = 0.0f;

        //for each sample
        for(int t = 0; t < numSamples; t++)
        {
              half2 offset = offsetData.offsets[t];

              //calculate the coordinates for this sample
              half2 sampleCoords = IN.UV + offset * centerPixel.a;

              //do the texture sampling for this sample
              half4 samplePixel = tex2D(ColorSamp, sampleCoords);
              half sampleDepth = tex2D(DepthSamp, sampleCoords);

              //--------------------------------------------------------------------------
              //Prevent focused foreground objects from bleeding onto blurry backgrounds
              //but allow focused background objects to bleed onto blurry foregrounds
              //--------------------------------------------------------------------------
              half weight = sampleDepth < centerDepth ? samplePixel.a * bleedingMult : 1.0f;
              weight = (centerPixel.a > samplePixel.a + bleedingBias) ? weight : 1.0f;
              weight = saturate(weight);
              //--------------------------------------------------------------------------

              //add this sample to the weighted average
              color += samplePixel * weight;
              totalWeight += weight;
       }

       //return the weighted average
       return color / totalWeight;
}
```

The result of the horizontal blur should look something like this:

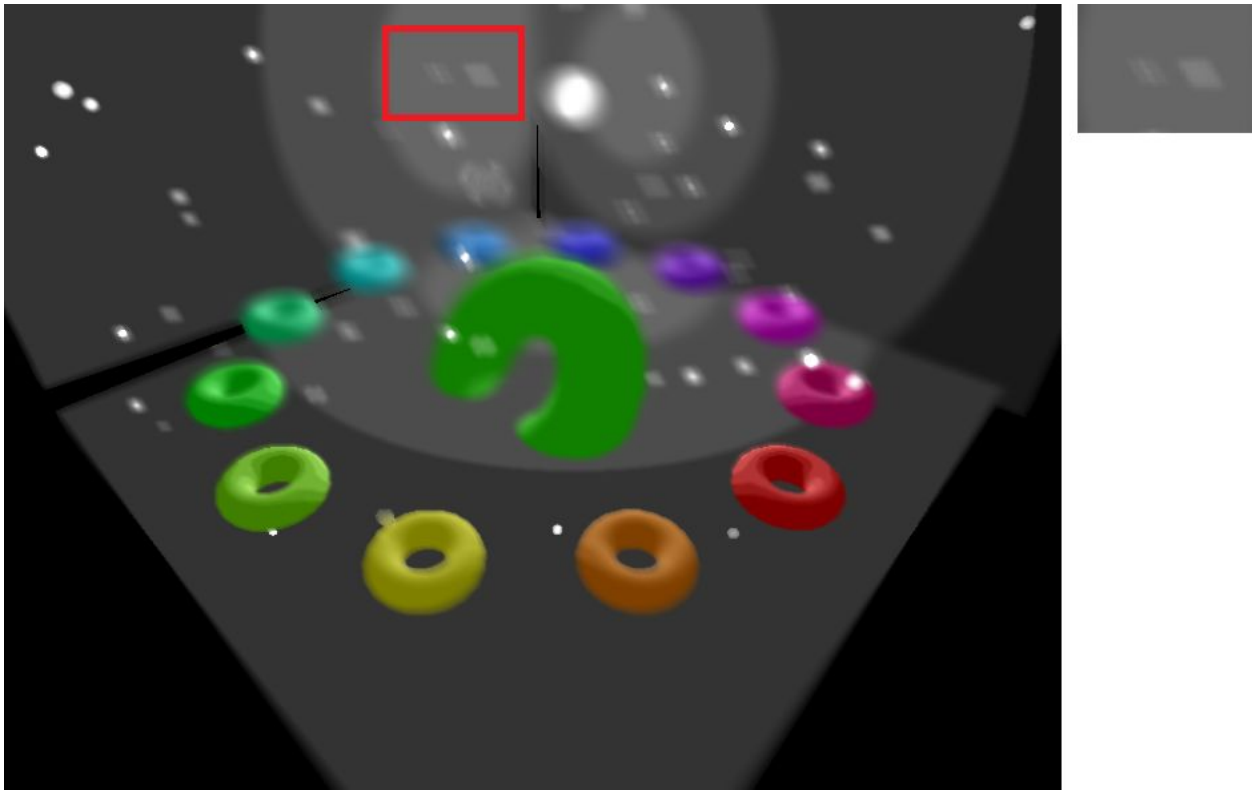


Notice that the points have been “smeared” horizontally.

Part 4: Diagonal Bokeh Pass A

So now we have written the bokeh_f.glsl shader, if your horizontal blur looks similar to the picture above you should be able to implement this pass very easily. All you need to do is use the exact same logic and shader as above, but this time use an angle of 120 and instead of blurring the scene texture, you are blurring the **horizontally blurred image.** Remember to convert to radians before you send the angle to the shader.

The resulting image should look something like this:

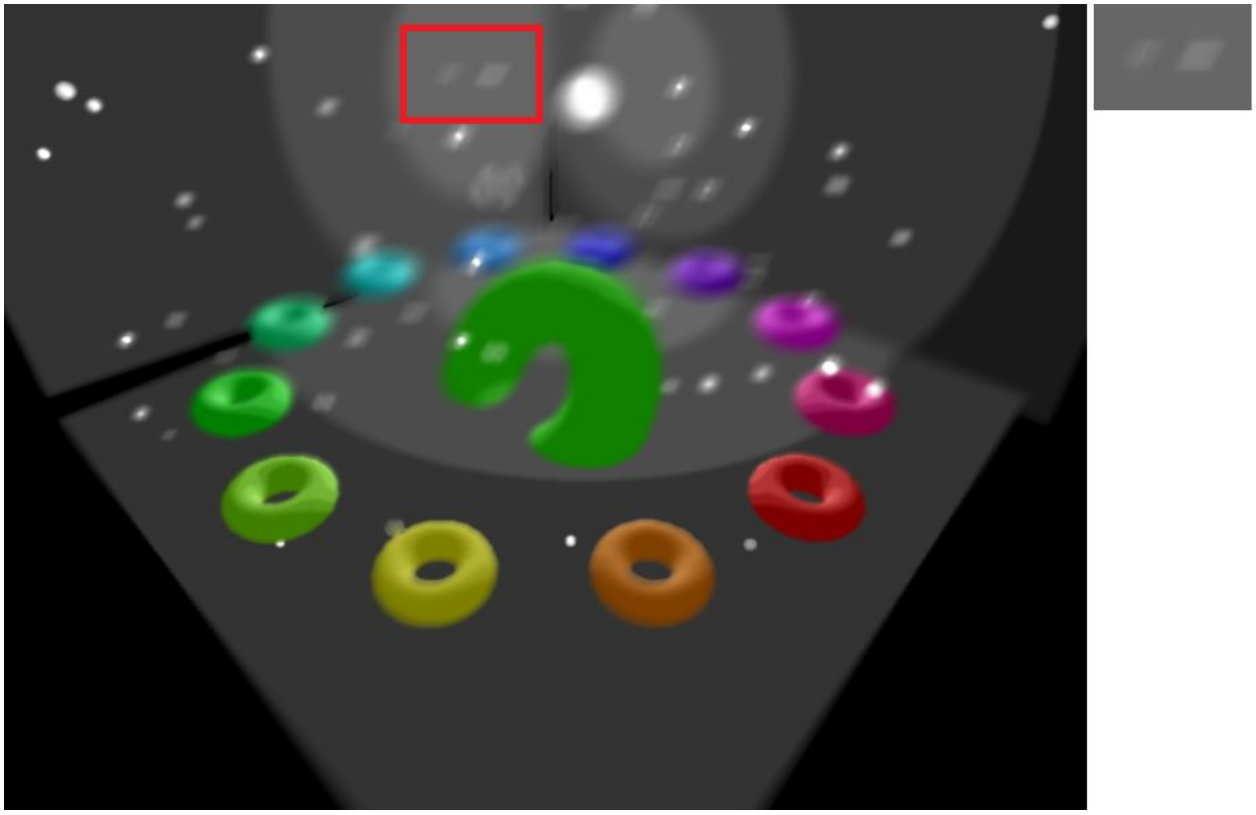


Notice that the horizontal blurs have been smeared diagonally.

Part 5: Diagonal Bokeh Pass B

This pass is exactly the same as the one above but this time use an angle of -120!

The resulting image should look like this:



Notice that the image looks very similar to the one above but this time the horizontal lines are smeared the other way.

Part 6: Compositing

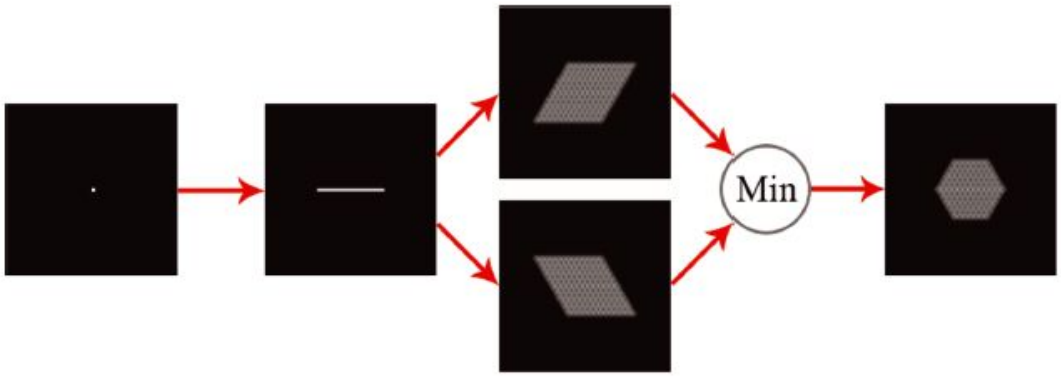
This is the final step, for this step you need to:

1. Create the horizontal Bokeh image (pass 3)
2. Create the first diagonal Bokeh image (pass 4)
3. Create the second diagonal Bokeh image (pass 5)
4. Bind the two diagonal Bokeh images (from steps 2 and 3)

In the composite shader implement the following logic:

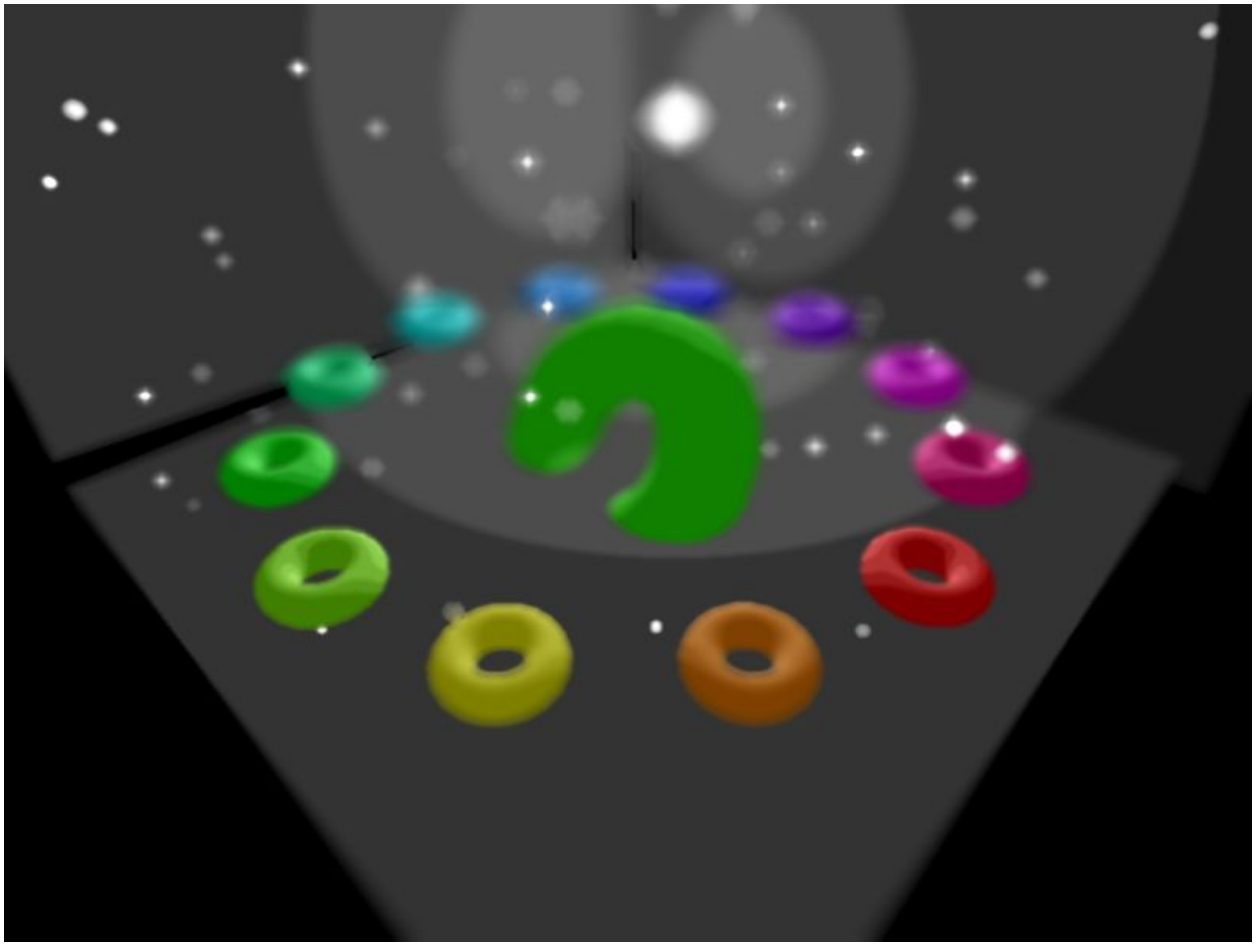
- Sample from Bokeh FBO A
- Sample from Bokeh FBO B
- Set the FragColor to: min(FBO A, FBO B)

Visually the algorithm is doing this:



(a)

The resulting image should look like this:



Grading

You are being graded on functionality and correctness.

IF YOUR CODE DOES NOT COMPILE YOU GET A ZERO FOR THE CODING PORTIONS OF THE LAB!!!

Initialization (/2)
- Did you create your frame buffer objects correctly?

Quality of horizontal blur pass (/4)
- Does your horizontal blur pass look correct?

Quality of diagonal blur passes (/4)
- Do your diagonal blur passes look correct?

Quality of Depth of Field effect (/4)
- Does your final composited DoF effect look correct?
- The final result should have **noticeable** Bokehs and it should be obvious which pixels are in focus.

Lab report with all explanations and requirements (/4)
- What is an aperture? What happens if you make the value larger? What happens if you make the value smaller? Try playing around with the aperture value used when calculating blurriness and take screenshots of some of the values you try. (/4)

Project directory submitted and is clean (/2)
- Do not submit the intermediate directory and the .db file. If your submission is extraordinarily large it may not be marked and you will receive a grade of 0.

****Project compiles and runs without error****
- IF YOUR CODE DOES NOT COMPILE (SHADERS TOO) YOU GET A ZERO FOR THE RELEVANT SECTIONS