Effects and Post-processing

Peder Bergebakken Sundt

My goals

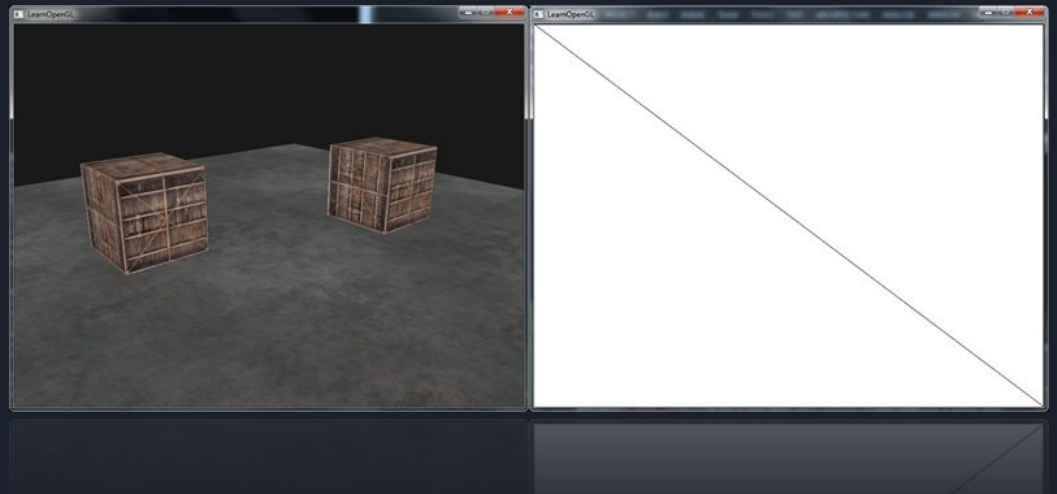
- Create one absolute banger of a scene in OpenGl
- Trying my hand on implementing as many effects as i possibly can:
  - Phong lighting, displacement mapping, reflections, fog, vignettes, chromatic aberration, depth of field and grain.
- Making it interesting: post-processing effects!

What is post processing?

- Post processing effects are effects you do, well..., in post!
  - This is *after* you have rendered your scene.
- Why do effects in post?
  - You can use the color and depth values of neighboring pixels!

Doing post-processing in OpenGl

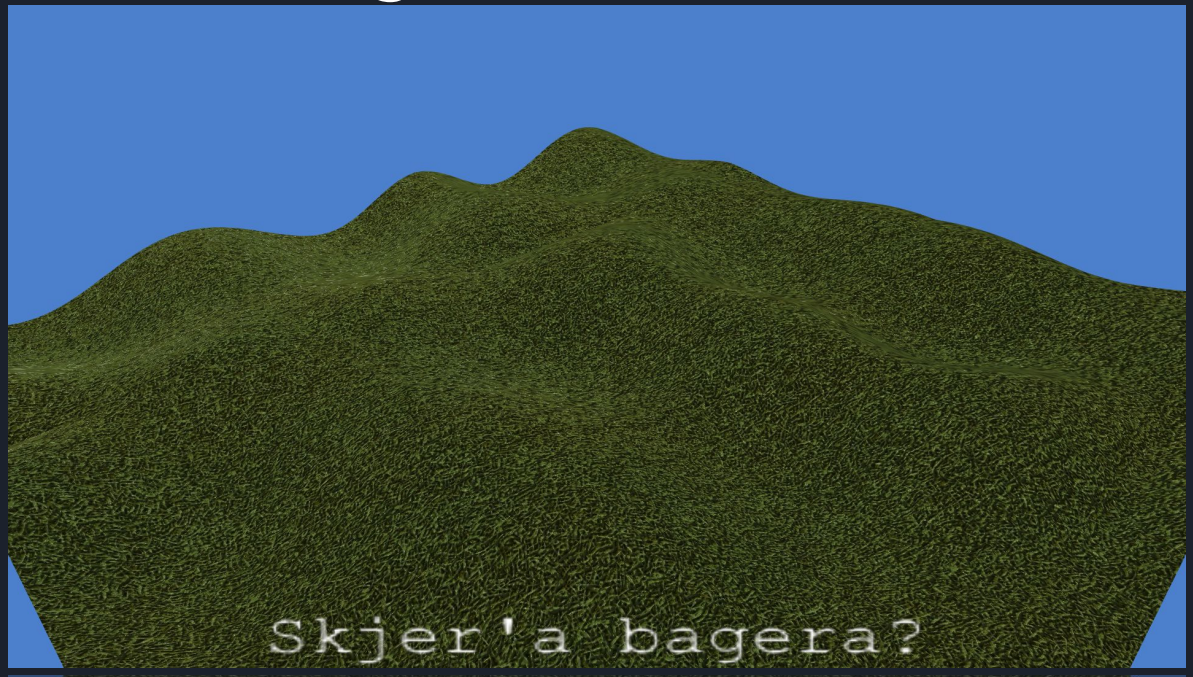
- OpenGl doesn't give us post-processing tools for free.
- ..but we do have framebuffers!
  - Framebuffers are what you'd use whenever you want to render your scene to a texture.
  - This is often used when computing shadows maps or real-time reflections
  - It's (probably) how the portals from Portal are made as well.
- We simply render the *whole* scene to a framebuffer instead of to the window.
- Then we render this framebuffer as a texture on a quad which covers the window.
- This allows us to apply a separate shader to this second rendering stage:



The scene I’m making

- A simple car model moving across a scrolling field of grass with, some tufts of grass and some trees spread about.
- There will be a live demo of it
- Yes, i did forget i had the ‘*Skjer’a bagera*?’ there throughout the whole development.

The field of grass

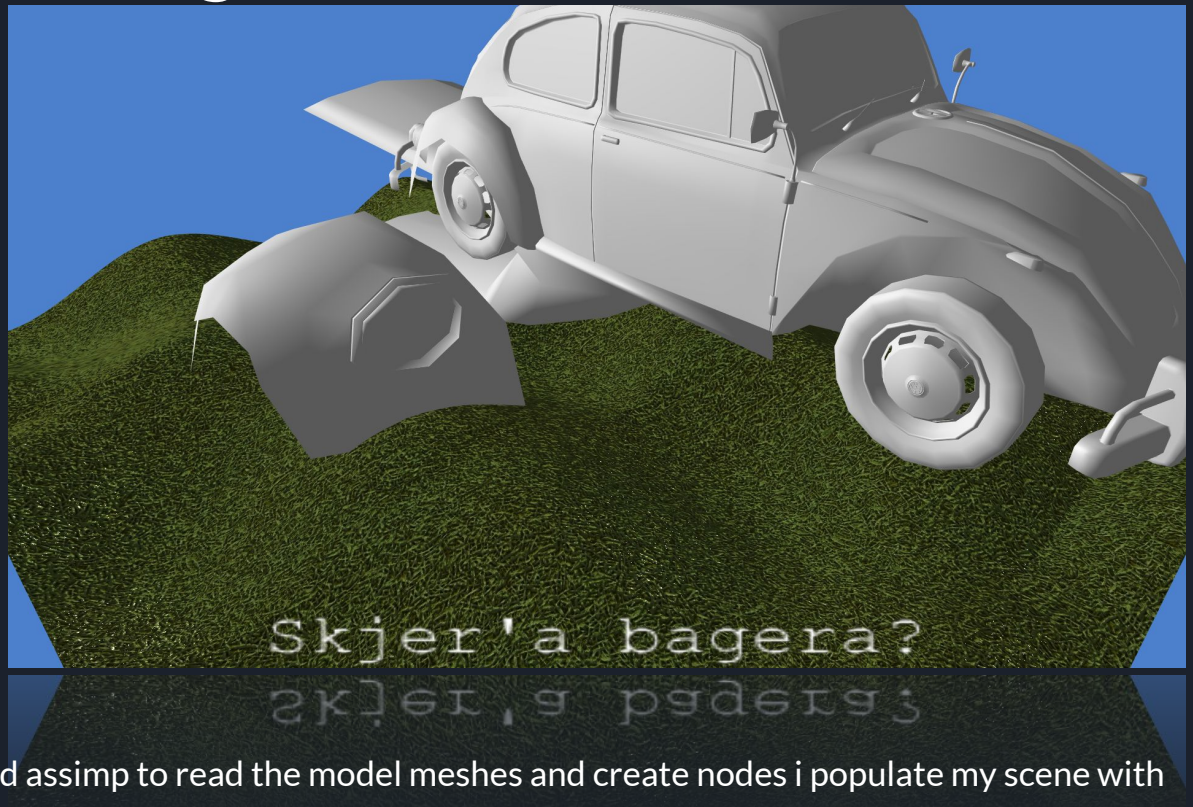


- I create a flat field mesh of 100x100 vertices, and then apply a displacement map in in the vertex shader. I scroll the field by adding a UV offset uniform to the UV positions.

The slope of the displacement

- The slope of the displacement map was used in the TBN matrix to have the normals rotate along wit the hills.

Loading the models



- I used assimp to read the model meshes and create nodes i populate my scene with

Applying transformations

- To apply the transformations, i decomposed the transformation matrix into position, rotation and scale. I then had to correct it for my coordinate system where z points skywards

Applying color

- To apply color, i had to implement a material system. Then i could load the materials with assimp and apply them to the nodes

Adding reflection

- The model used a metallic effect, but i instead opted to implement reflection.
- It reflects the camera->fragment vector along the surface normal and maps this reflection into a spherical reflection map

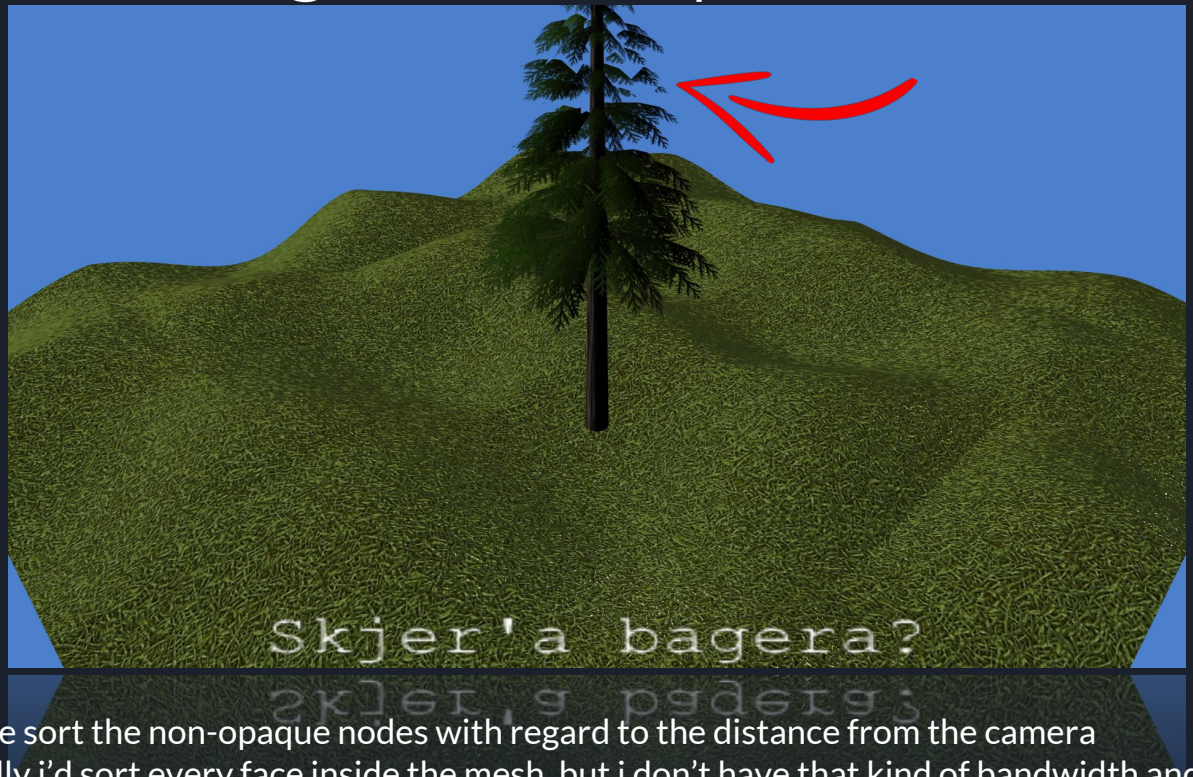
Reading textures

- The model loader doesn’t read textures yet.

Reading textures?

- The textures have now been read, the transparency however doesn’t work well.

Rendering the transparent nodes



- I here sort the non-opaque nodes with regard to the distance from the camera
- Ideally i'd sort every face inside the mesh, but i don't have that kind of bandwidth and computation capacity

Cheating the depth test

- The quick fix i landed on was to set the depth buffer to be in read-only mode after having rendered all the opaque objects first.
- I then render the transparent objects in sorted order, back to front.

Rim lights!

- Rim lights are often used in cinematography and games to make objects ‘pop’ more, and separate then from the backgrounds
- The object simply lights up the more it faces away from the camera

Rim lights

- This was implemented in the scene shader, controlled by uniform values set per node:

```
color * clamp((dot(normalize(vertex), normal) + strength) / strength, 0, 1); // in MV space
```

Post-processing: Creating the framebuffer

The framebuffer is simply an empty texture stored in memory. We use mirrored repeat to make blurs behave nicely along the edges. We also have to create a depth buffer.

```
if (first) glGenFramebuffers(1, &framebufferID);
glBindFramebuffer(GL_FRAMEBUFFER, framebufferID);

if (first) glGenTextures(1, &framebufferTextureID);
glBindTexture(GL_TEXTURE_2D, framebufferTextureID);

// Give an empty image to OpenGL ( the last "0" )
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, windowWidth, windowHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);

if (first) glGenTextures(1, &framebufferDepthTextureID);
glBindTexture(GL_TEXTURE_2D, framebufferDepthTextureID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24, windowWidth, windowHeight, 0, GL_DEPTH_COMPONENT, GL_FLOAT, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

Post-processing: Binding the framebuffer

Now we have to bind the depth buffer to our depth buffer texture, and bind our color buffer as attachment number 0.

```
if (first) glGenRenderbuffers(1, &framebufferDepthBufferID);
glBindRenderbuffer(GL_RENDERBUFFER, framebufferDepthBufferID);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, windowWidth, windowHeight);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, framebufferDepthBufferID);

// Set "framebufferTextureID" as our colour attachement #0
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, framebufferTextureID, 0);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, framebufferDepthTextureID, 0);

// Set the list of draw buffers.
GLenum drawBuffers[] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, drawBuffers);

glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ONE_MINUS_SRC_ALPHA);

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Post-processing: Rendering to the framebuffer

First we have to bind the framebuffer as our render target:

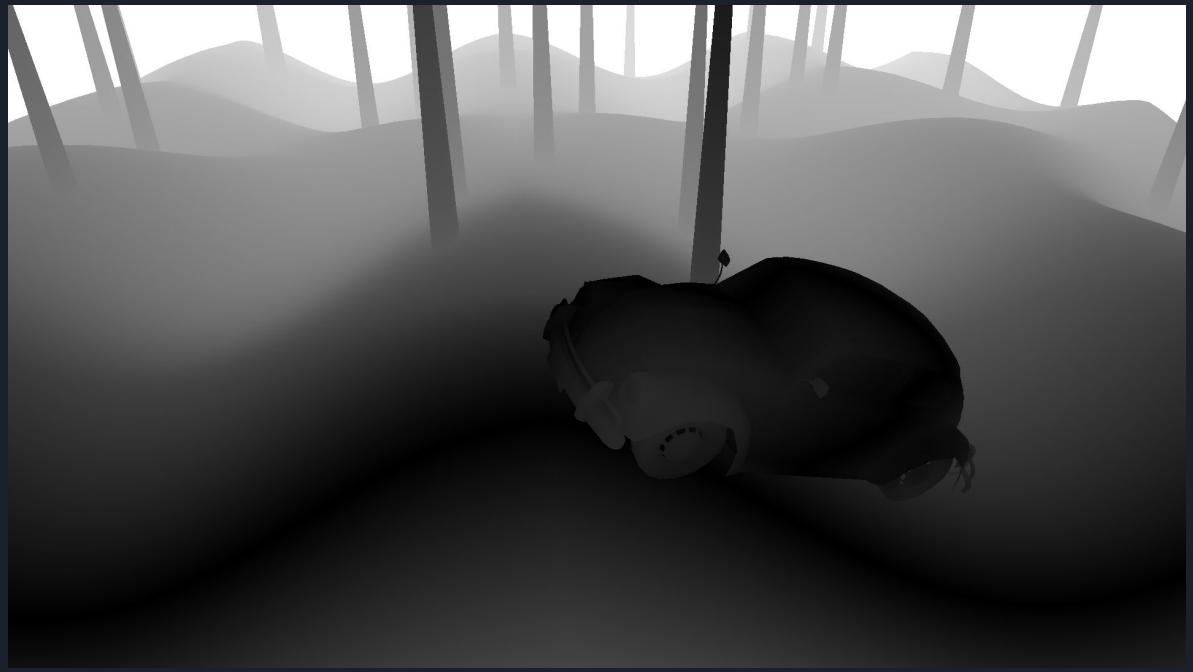
```
// render to internal buffer
glBindFramebuffer(GL_FRAMEBUFFER, framebufferID);
glViewport(0, 0, windowWidth, windowHeight);

// Clear colour and depth buffers
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

renderNode(rootNode);
```

Then we have to select color attachment as our output in the fragment shader:

```
layout(location = 0) out vec4 color_out;
```

Time to make some post-processing effects!

Point of focus

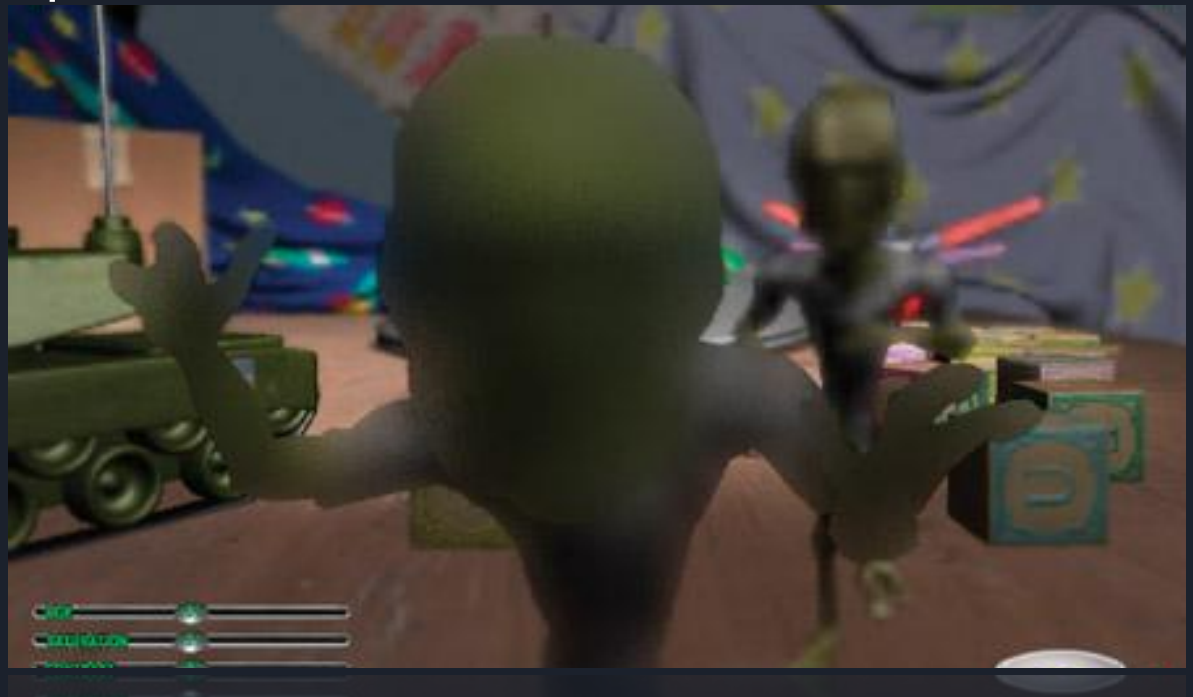


- My ‘point of focus’ is the car in my scene. Here i simply transform the depth buffer to make it linear, then i center it around the depth of the car. 0 being in focus and 1 when moving away from it.
- Note: the grass and leaves are gone since they were rendered while in read-only mode.

Depth of field

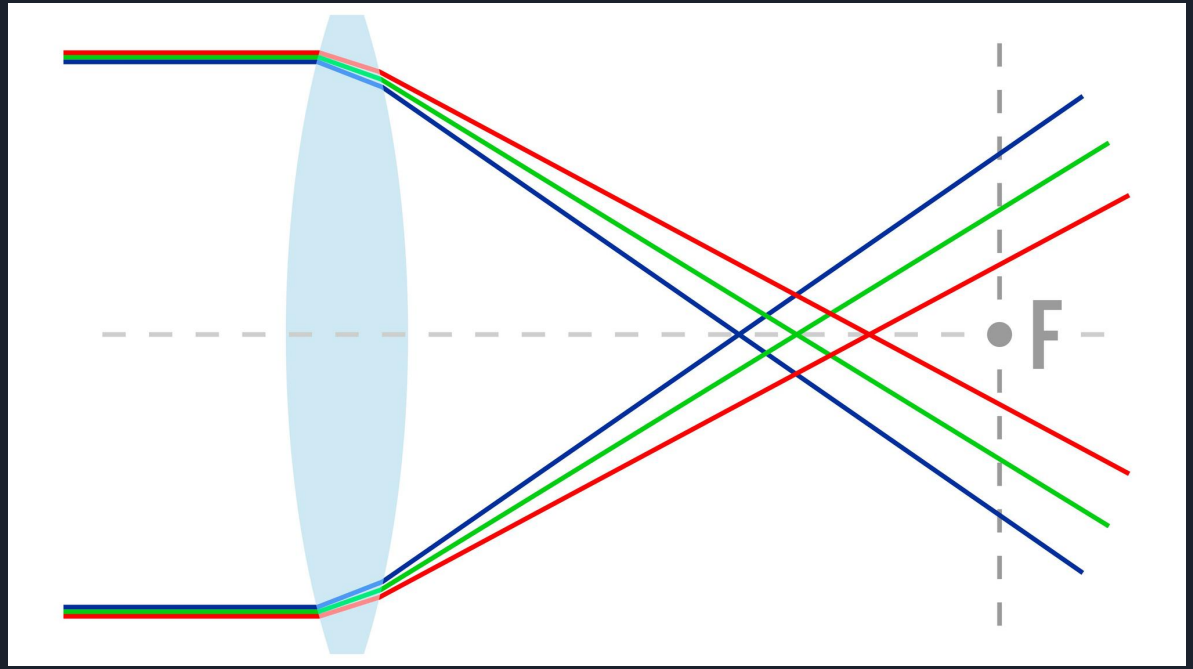
- My implementation is simple: Blur each pixel with the range set by the point of focus.
- Here we can *kind of* see the effects of on the trees. The darkness hides it, kinda...

Depth of field: Weaknesses



- Since my implementation is only a blur with the radius controlled by the depth of the centermost fragment, i don’t get any defocusing of the edges. This could easily be solved always blurring with max radius and instead the contribution of each pixel by using the focus, but i don’t need that added complexity to make my specific scene look good.

Chromatic aberration: what's that?



- Chromatic aberration is the effect of different wavelengths of light refracting differently. This can often be seen when using shitty lenses on your camera.
- This, combined with depth of field create a macro-lens feel
- Is dependent on the distance from the center ad the distance from the point of focus.

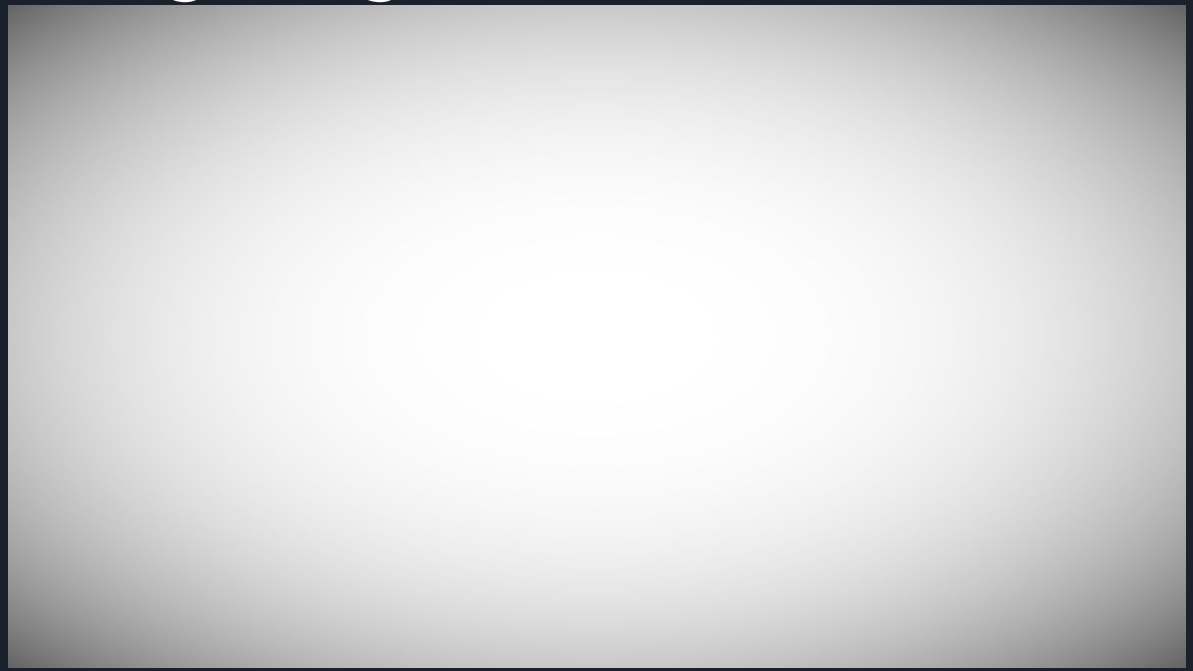
DoF and chromatic aberration shader

```
int radius = int(5*z);
vec3 color = vec3(0);
for (int x = -radius; x <= radius; x++)
for (int y = -radius; y <= radius; y++){
    vec2 p = UV + x*dx + y*dy;
    color.r += texture(framebuffer, (p-0.5)*(1+z*chomatic_abberation_r) + 0.5).r;
    color.g += texture(framebuffer, (p-0.5)*(1+z*chomatic_abberation_g) + 0.5).g;
    color.b += texture(framebuffer, (p-0.5)*(1+z*chomatic_abberation_b) + 0.5).b;
}
color /= pow(2*radius+1, 2);
```

- Z is the focus factor

Chromatic aberration: result

Adding a vignette



- Darkening of the image the close to the edge you get. Created using the euclidean distance
- Here applied to a white framebuffer

Adding grain

- Grain is simply random noise. It's often added to the image for realism.
- I had to include my own random generator. It uses the UV coordinates and the time as the seed.
- I chose to add more grain to the out-of-focus areas, using the depth buffer

Fog

- Fog is added using the depth. I couldn’t do it in post, due to grass and leaves not showing up in the depth buffer, but adding fog in the scene shader is just as easy.
- Simply linearize the depth component of the MVP coordinates, then use this as the mixing factor between the original color and our fog color. I could play around with polynomial scaling of the depth, but linear fog is sufficient for my scene.

All done
Time for the demo!