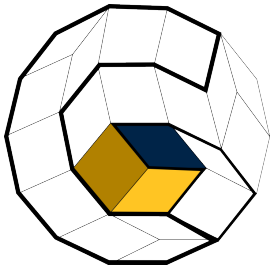
TU BRAUNSCHWEIG
Dipl. Inform. Martin Eisemann
Institut für Computergraphik
Lorenz Rogge (rogge@cg.tu-bs.de)

June 28, 2011



Echtzeit Computergraphik SS 2011
Assignment 9

In this exercise we want to simulate a depth-of-field effect, as it would be natural using a real camera. Please present your solution to this exercise on Monday, July 4th, 2011.

Note: The given framework has be rearranged to be more flexible. Two classes have been added using all the functionality you have implemented so far. The *Shader* Object handles all issues related to loading and using shader programs. A class *Material* hab been added as well, which allow to control material properties disjoint from a specific object, that has to be rendered. The new structure is as follows:

- Materials use Shaders to render their material effect properly. Textures can be used besides usual material properties like diffuse, specular, emissive, and ambient color components. (Phong model)
- MeshObj objects have a material attached to them, that is been used for rendering. So whenever the method `MeshObj::render()` is executed, internally, the material and its attached shader program will be used for rendering.
- Shaders can be simply constructed by passing the source code files to a Shaders constructor. The shader may then be attached to a material or used a stand-alone shader withing your code.

9.1 Rendering into textures using Frame Buffer Objects (20 Points)

To create the desired depth-of-field effect, where distant or very close objects are blurred, while objects in focus remain sharp, we could render a lot of sample images from different positions by moving our OpenGL pinhole camera all over the sensor area we want to simulate. This of course would be very accurate (with respect to the number of samples taken) but also *very* slow. So instead of resampling the whole scene, we capture it once and blur the rendered image afterward, depending on how deep a pixel is compared to our desired focus depth.
To be able to postprocess a rendered image with shaders, we need to render our scene into a OpenGL texture instead directly rendering it onto screen.
So in this first task, add the functionality of FrameBufferObjects to your framework. Implement the method `initFBO` within `Ex09.cpp` to initialize and prepare a FrameBufferObject.

- Generate OpenGL texture objects that will be attached to your framebuffer. You can use these for simple color textures, but also to save the rendered depth-buffer, which is cruicial in the following task 9.2. By using `GL_DEPTH_COMPONENT` as data type of your texture, you can store depth values just like storing any other image.
- To create a FrameBufferObject, call `glGenFramebuffers`.
- Having a valid FBO, bind it and attach all desired textures to it using `glFramebufferTexture2D`. With the second parameter you may control to which color or data attachment this texture belongs. For example `GL_COLOR_ATTACHMENT0` attaches the given texture object to the first color unit of this framebuffer which is written by `gl_FragColor` in your shader. `gl_FragData[1]` however will try to write a buffer attached to `GL_COLOR_ATTACHMENT1`. The parameter `GL_DEPTH_ATTACHMENT` may be used, when writing the depth buffer into a texture.

9.2 Depth-Of-Field using local image blur (20 Points)

To create the illusion of depth-of-field, we want to blur the rendered image dependent on the depth of a given pixel. The amount of blur increases with the difference in depth compared to a given focus-distance. To pixels with a great depth difference must be filtered with a much larger blur kernel as pixels close to the focus-distance.
To speed up the image filtering, we do not compute the mean of a rectangular image region for each pixel, but instead use two seperate render passes. The first render pass only blurs in horizontal direction while a second render pass blurs the previously processed image in vertical direction. Using a filter kernel of 5 by 5 pixels this results in only 10 computations per pixel instead of 25.
To create your blurred image, implement the following:

- Render the scene to a FrameBufferObject, while rendering the color values to the first color attachment and the depth buffer to the depth attachment of this FBO.
- Create and enable a shader blurring a given input image using a given depth map. These can be passed to the shader as textures, since we have them rendered to our FBO.
- As a first render pass, render a screen filling quad with the two given textures and filter the color image in horizontal direction. Pass a blur magnitude and a desired focus depth (`blurStrength` and `focusDepth`; both already given in the code) to the shader an compute the blur region for each pixel. Simply compute the difference in depth for each pixel and increase the sample region when filtering the image. In this render pass use the FBO *again* and render the result to a texture.
- Finally render a screen filling quad using the blur shader in vertical direction (just as the last render pass) and render the result to screen (`glBindFramebuffer(GL_FRAMEBUFFER, 0)` disables al active FBOs and switches back to the standard output buffer - the screen).

When you are done, you should see a result like in figure 1.

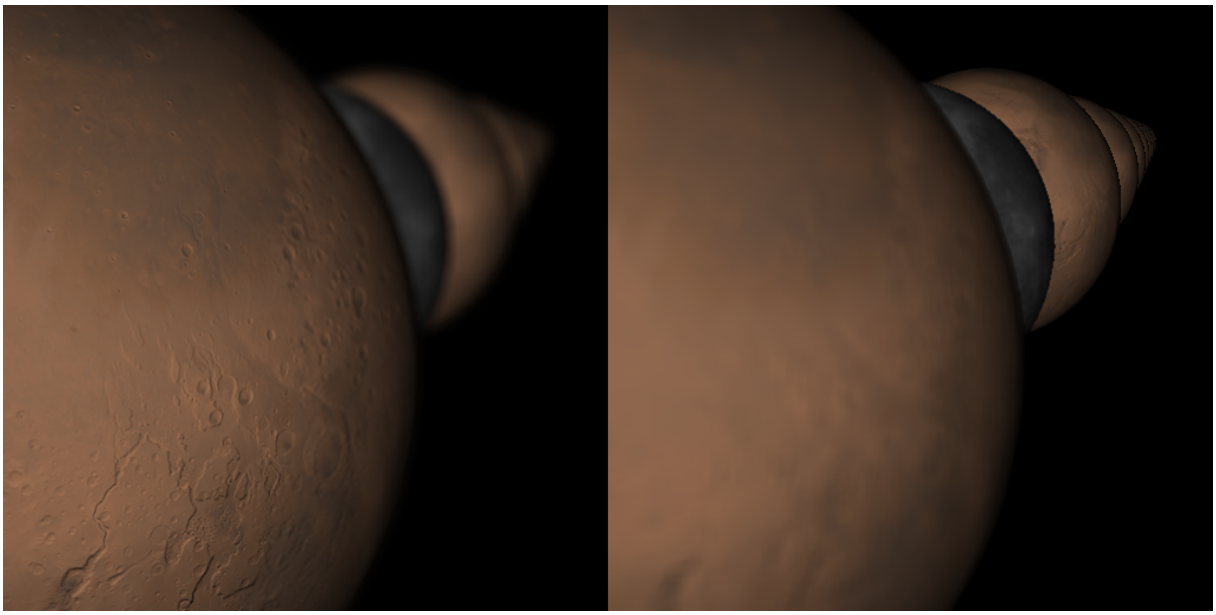


Figure 1: Image is blurred with increasing depth.

To evaluate your implementation, the framework already allows to control both focus depth and blur magnitude (`focusDepth` resp. `blurStrength`) with the keys `+/-` resp. `1`, `2`, and `3`.