TU BRAUNSCHWEIG
Dr. Georgia Albuquerque
Institut für Computergraphik
Matthias Überheide

June 19, 2018

Real-time Computer Graphics, Summer 2018
Assignment 8

Present your solution to this exercise on Thursday, June 28, 2018.

The exercises are going to take place in the CIP pool, room G40 in Mühlenpfordstraße 23. Please make sure your solutions compile and run on the CIP pool computers. Note that you need a y-number, which can be obtained at the Gauß-IT-Zentrum, to use the computers. If for some reason you are not able to attend the exercise, you may send your solution to ecg@cg.cs.tu-bs.de instead.

This exercise and the corresponding version of the framework can be found on the lectures website (`http://www.cg.cs.tu-bs.de/teaching/lectures/ss18/ecg/`).

Theoretical Tasks

8.1 Deferred Shading Efficiency (50 Points)

Consider a scenario where deferred shading is used with the intention to reduce rendertime. The intended shading needs to access a texture three times. For the deffered shading 6 color attachments are required but the first pass does no longer need to access any textures. Compute the necessary average overdraw for which the deferred shading increases efficiency. Use the number of texture access operations as a metric.

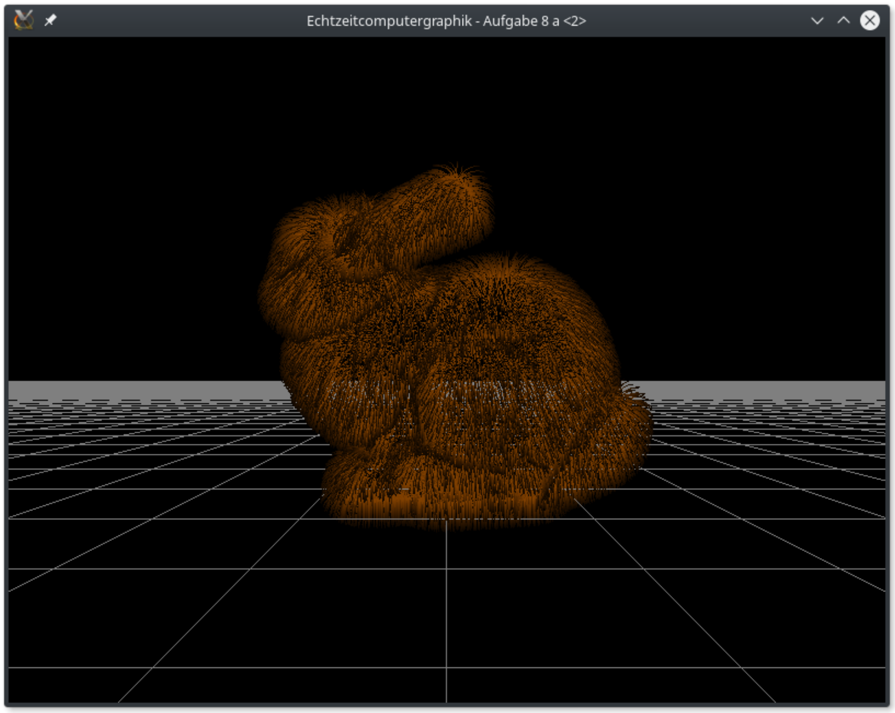
8.2 Short Presentation (Extra credit 10 Points)

Prepare a 5min talk about an interesting topic related to the current lecture. Send your topic proposal via email to ecg@cg.cs.tu-bs.de at least 24 hours before the presentation.

Practical Tasks

8.3 Rendering line-based hairs (30 Points)

In the geometry shader `shader/hair_lines.geom` hair has to be rendered by creating line strips. First, define the number of maximum output vertices. This sets the upper limit of output vertices per geometry shader call on the GPU. Then define the shader layout. The input are triangles whereby the output should be line strips.
Next, add the uniforms for defining hair properties. Then set additional input and output variables of the shader. Input should be the per-vertex normals. The output is a single color per vertex.
Now implement the main function of the shader. Each hair should start at the center of the triangle into normal direction. The precision is controlled by the amount of segments. Add a simple effect so that each hair seems to be affected by gravity. This is realized by adding a gravity value on the normal for each segment of the hair. For shading in this task you can simply compute some gradient to the vertex colors based on the material color until it looks hairy and good enough.



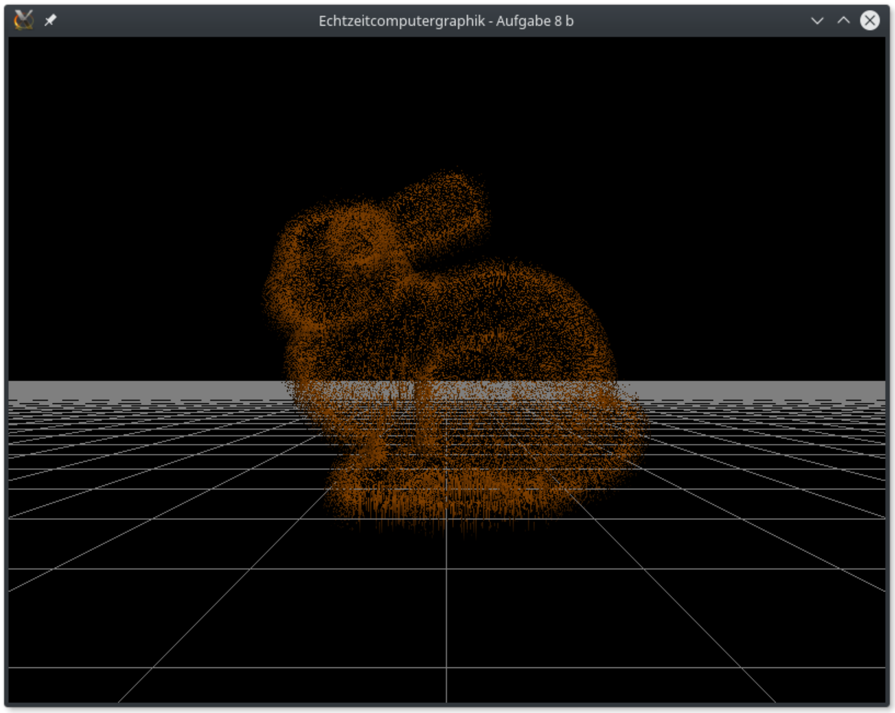
8.4 Rendering triangle-based hair (30 Points)

Now it is time to enhance the realism of the hair. Use the second geometry shader program (`shader/hair.geom`) to render the hair using triangle strips instead of line strips. Adapt the shader layout appropriately and the vertex computation part in the shader. To get a hair-like triangle strip, start at the center of the input triangle shift the center by half of the hair width, which is given by a define in the shader, and span the triangle strip into normal direction. When stepping further along the hair towards the tip of the hair reduce the width for each segment until reaching the tip (zero width). Pay attention to the direction of the triangles. Each triangle should always be kept camera-oriented to remain visible. Therefore, all computations should be performed in view-space. For shading and gravity use the same approach as in the previous task.

8.5 FBO Creation and Deferred Shading (60 Points) *Optional*

You may choose to skip this task if you prefer.
The most important thing for deferred shading is to create and enable a frame buffer object. We also need to attach several textures to this render buffer.

The FBO in this exercise needs to be able to handle depth information, since it is needed when rendering the scene the first time. We don't need to access this depth information by ourselves, but the FBO needs to have an attachment to the depth component.
Complete the code in `gl/OGLFramebufferRenderPass.cpp` to get a working render pass. Start by implementing the `setupBuffers()` and `renderCall()` methods. You need to create a framebuffer, bind it and request attachments for which textures have to be made available.
Implement a deferred normal mapping with `shader/deferred/pass1` and `shader/deferred/pass2`. Deferred shading decouples the actual rendering of the geometry of the shading process for every pixel. In very complex scenes with many overlapping objects, it is not neccessary to compute the shading for pixels, that will be covered by another object anyways. We cannot really control the order, that the objects will be rendered in, so it might happen, that the visible pixel is rendered last after several other rendering of now occluded objects. To reduce this overhead, deferred shading renders the geoemetry first, storing only the important vertex (or pixel) attributes in separate texture layers. In a second rendering pass, the previously created textures will be mapped to a simple screen-filling quad. Instead



of texturing this quad directly with the given textures, the per-pixel-data will be used to evaluate the shading computation of the first render pass. Now only the truely visible pixels are processed and shaded. This speeds up the rendering a lot.

The given exercise uses normal mapping and one color texture layer for object shading. So the per-pixel-information that we need for the shading in the second pass will be:

- The position of the pixel in camera space.
- The surface normal of this pixel. (Since we are using normal mapping, we store the already modified surface normal here. So the normal map has to be accessed in the *first* render pass).
- The texture coordinate used by this pixel.

Implement the shader used for the first rendering pass in `shader/deferred/pass1.vert` and `shader/deferred/pass1.frag`. This shader has the original vertex data a input, transforms it as usual according to the modelview and perspective transform. Since we need to compute the modified surface normal in this pass, too, transform the provided normal, binormal and tangent information in the vertex shader using the normal matrix and pass it to the fragment shader. In this shader, compute the tangent space matrix as in the last exercise, but now transform the normal given in the normal map into camera space and not the other way around. Use the inverse of the tangent space matrix for this conversion.
Now, that you have all the data needed, pass the values for pixel position, normal and texture coordinate to the color outputs of your shader.
The second rendering pass finally does the shading of the pixels. Implement the missing parts in `shader/deferred/pass2.frag`.
The rendering will be triggered by a screen filling quad, so that ever screen pixel will be evaluated exactly once. This is already prepared.
In the fragment stage, get all the needed information from the three textures of the last rendering pass (position, normal, texture coordinate) and use them to compute the shading as usual.
Use the texture coordinate provided by the attribute texture to access the surface texture at the correct position. Combine the material shading with this texture value to shade your final pixel.

One problem still remains: When you render your screen filling quad, you will need to find a

way to decide, whether the pixel you are processing was covered by geometry in the first rendering pass or not. Find a way, to evaluate this. If a pixel has not been covered by any geometry, you can simply `discard` it in your fragment shader of the second pass.
You can explore the same scene using `ecg_ex08_c_nodef` which does not use deferred shading (this might be slow on your computer).

8.6 Coding (Extra credit 10 Points)

Present one of the following:

- a bug in the framework
- a helpful test case to solve an exercise
- the implementation of an additional feature
- a nice demo using the current framework

Note that a short documentation is required and code has to be handed in. Please report bugs immediately (via email) so they can be fixed as soon as possible.