UNIVERSITÄT DES SAARLANDES
Dr.-Ing. Hendrik P.A. Lensch
Max Planck Institut Informatik
Art Tevs (tevs@mpi-inf.mpg.de)
Boris Ajdin (bajdin@mpi-inf.mpg.de)
Matthias Hullin (hullin@mpi-inf.mpg.de)

26. November 2007

Computer Graphics I
Assignment 5

Submission deadline for the exercises: Monday, 3th December 2007
Rule: Written solutions have to be submitted in the lecture room before the lecture. Solutions of practical exercises have to be submitted until **8:30am**, hence before the lecture!

5.1 Fourier Transformation (15 Points)

Show that the Fourier transformation of the box function $B(x)$ is a *sinc* type function. The sinc function is defined as $sinc(x) = \frac{sin(\pi x)}{\pi x}$ and a definition of the Fourier transform can be found in Exercise 5.5.

$$B(x) = \begin{cases} 0 & \texttt{for } x \leq -1 \\ 1 & \texttt{for } -1 < x < 1 \\ 0 & \texttt{for } 1 \leq x \end{cases}$$

5.2 Procedural Shaders - Clouds (30 Points)

In this exercise we will implement a shader which is able to compute a cloud like structure. Look for more information about procedural shaders or procedural textures on the internet.

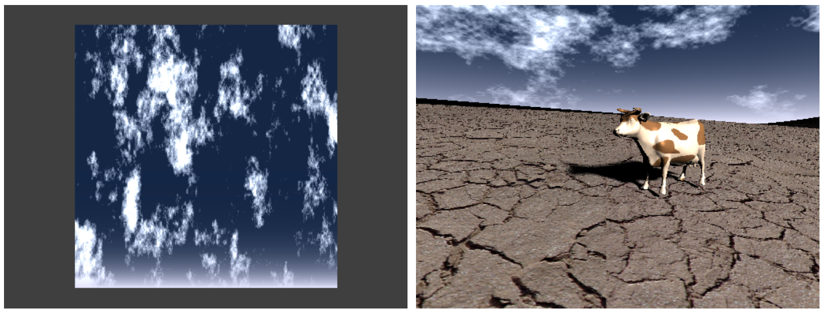


Figure 1: Left: Cloud structure generated by perlin noise. Right: Cloud shader applied on the skydome object.

For simulating clouds we incorporate the perlin noise algorithm which is capable of creating correlated noise. Since the perlin noise depends on a random function, it is not necessary to create exactly the same results as in the provided screenshots. See figure 1 as one example how your results should look like. As atmosphere color value you can use $color = (0.08, 0.15, 0.27)$.

a) Implement `PerlinNoise2D` in `stdafx.cxx`. The function should return a sampled value of a 2D perlin noise function. (20 Points)

b) Implement the `CloudShader::Shade()` method. Incorporate the 2D perlin noise to generate nice looking noise structures. (5 Points)

c) To emulate atmospheric scattering effects we linearly interpolate the computed color value with white color based on the v-coordinate of the texture coordinates, i.e. $(1-v)^{12}$. (5 Points)

d) Add *skydome.obj* into your scene and apply the `CloudShader` on it.

5.3 Alpha Blending (15 Points)

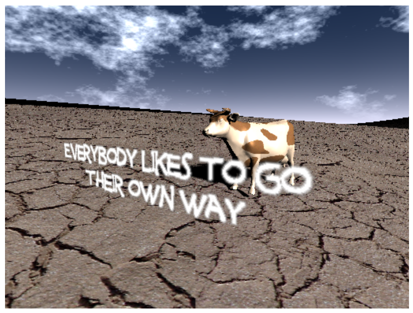
Until now we have handled textures with only three channels per pixel, i.e. RGB. However there are also images/textures with 4 channels per pixel, i.e. RGBA. The fourth channel, called *alpha-channel*, could store transparency information of the according pixel. In most of the cases alpha value is used as a blending factor. There is a lot of possibilities of how to combine the source color c_1 and destination color c_2. For the sake of simplicity we will limit ourself to the following formulation:

$$c.rgba = lerp(c_2.rgba, c_1.rgba, c_1.a).$$

This blending mode is equivalent to the OpenGL's blending mode specified by `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.
In this exercise you have to implement missing parts of the `FlatTransparentShader.hxx` which should support alpha blending as presented here.

- Combine all assigned textures as shown in the previous equation.
- Think about what is the main property of transparent objects. What should you do with your rays to make object transparent? Implement the `FlatTransparentShader::Shade()` method.
- Add new object *mesh/wisdom.obj* into your scene. Apply the `FlatTransparentShader` on the newly added object. Add the *textures/wisdom.png* to the shader. Setup linear interpolation mode on this texture. Call `wisdom->setCastShadows(false);` to disable shadow casting of the wisdom quad, since we currently do not support shadows of transparent objects.

If your implementation is correct you should get the following (i.e. `result_5_3.png`):



Note: The implementation of `Image.cxx` and `Texture2D.hxx` were changed to support RGBA textures.

5.4 Supersampling (40 Points)

A pixel actually corresponds to a square area. Currently you are sampling the pixels only at their center, which lead to aliasing. As you have learned in the lecture, the most simple way for removing aliasing artifacts from your image is *supersampling*, i.e. to shoot more than one ray per pixel. The three most frequently used supersampling strategies are:

Regular Sampling: The Pixel is subdivided into $n = m \times m$ equally sized regions, which are sampled in the middle:

$$samplepos = \left(\frac{i+\frac{1}{2}}{m}, \frac{j+\frac{1}{2}}{m}\right)_{i,j=0}^{m-1}.$$

Random Sampling: The Pixel is sampled by n randomly placed samples $\xi_i \in [0,1)$:

$$samplepos = (\xi_{i,1}, \xi_{i,2})_{i=0}^{n-1}.$$

Stratified Sampling: Stratified sampling is a combination of regular and random sampling. One sample is randomly placed in each of the $n = m \times m$ regions with $\xi_i, \xi_j \in [0,1)$:

$$samplepos = \left(\frac{i+\xi_i}{m}, \frac{j+\xi_j}{m}\right)_{i,j=0}^{m-1}.$$

In this exercise your task is to implement these sampling strategies:
In the framework you can find an abstract base class `SampleGenerator` with one single virtual method `void SampleGenerator::GetSamples(int n, float *u, float *v, float *weight)` that is supposed to works as follows: `n` is the number of samples to be generated for a pixel. One sample consists of two coordinates (u,v) that specify a position on a pixel. The `n` samples generated are to be returned in the `u` and `v` arrays, where (u,v) should be in the domain $[0,1) \times [0,1)$. The weights for the individual samples should sum up to 1. Here, just use uniform weights with `weight[i]=1.0/n`.

a) In your main loop, produce `n` samples, and fire `n` rays through the pixel at the respective sample position. The resulting color values must be weighted by `weight[i]` and summed up yielding the final pixel result. Implement the `RenderSupersampledFrame()` in `MicroTrace.cxx`(5 Points)

b) Implement the `GetSamples`-method in `RegularSampleGenerator.hxx`, `RandomSampleGenerator.hxx`, and `StratifiedSampleGenerator.hxx`, which are derived classes from `SampleGenerator`. (20 points)

c) Implement a different sampling strategy as described here. Find a suitable strategy, e.g. PossonDisc, RotatedGrid, or any other which you like to use. (15 Points)

d) Compare results generated by different sampling strategies with the results generated without supersampling. Which differences can you see? Which one would you prefer?

To compare your implementation you can download the result images for this exercise from: `http://www.mpi-inf.mpg.de/departments/d4/teaching/ws2007-2008/cg/ex5/results.zip`

5.5 Duality of Multiplication and Convolution* (20 Points)

The convolution of a function $f(t)$ with a second function $g(t)$ is defined as:

$$(f \otimes g)(t) = \int_{-\infty}^{+\infty} f(\tau) \cdot g(t-\tau) d\tau$$

The multiplication of two function is defined as the point-wise multiplication:

$$(f \cdot g)(t) = f(t) \cdot g(t)$$

The transformation of a signal $f(x)$ to Fourier space is given by:

$$F(k) = \int_{-\infty}^{+\infty} f(x) \cdot e^{-2\pi ikx} dx$$

We call $\mathcal{F}$ the operator mapping f to Fourier space: $\mathcal{F}f = F$. Show that convolving in signal space is the same as multiplication in Fourier space:

$$\mathcal{F}[f \otimes g] = \mathcal{F}[f] \cdot \mathcal{F}[g]$$