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)

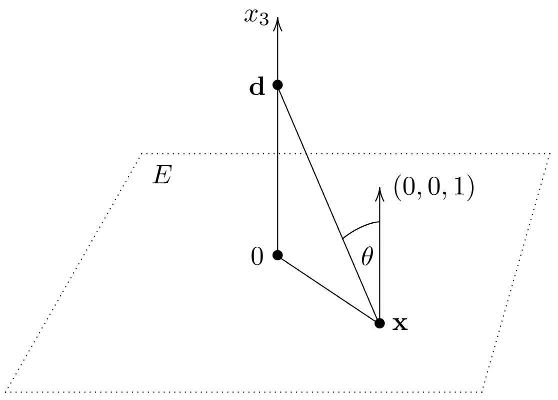
19. November 2007

Computer Graphics I
Assignment 4

Submission deadline for the exercises: Monday, 26th November 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!

4.1 Radiometry (10 + 10 Points)

A point light source with isotropic radiance is placed in $\mathbf{d} = (0, 0, d) \in \mathbb{R}^3, d > 0$. Let its power be $\Phi_S = 100W$. Consider the infinite plane E spanned by the x_1- and x_2-axis, defined by the condition $x_3 = 0$.



Your tasks are the following (**Note: (a) and (b) can be solved independently of each other, i.e. you may use the result for $I(r)$ given in (a) in order to start with the second part**):

a) Let $\mathbf{x} \in E$ be an infinitesimal patch with distance $r \geq 0$ to the origin. Show that the irradiance $I(r)$ at this point is given by

$$I(r) = \frac{\Phi_S \cos(\theta)}{4\pi(r^2 + d^2)},$$

where θ is the angle between the plane normal $(0, 0, 1)$ and the direction from $\mathbf{x}$ to $\mathbf{d}$.

b) Compute the radiant power Φ_E received by E by integrating the irradiance over E. The surface integral in polar coordinates is

$$\Phi_E = \int_0^{2\pi} \int_0^{\infty} I(r) r \, dr \, d\phi.$$

Does the result surprise you?

4.2 Interpolation (10 + 10 Points)

Interpolation is a method of computing new data points from a given discrete set of data points. *Linear interpolation* interpolates the missing values of a function with one variable $f : \mathbb{R} \to \mathbb{R}$ linearly for the domain interval $[x_0, x_1]$. For example: given two data points $f(2) = 5$ and $f(4) = 8$ (i.e. interval is $[2, 4]$) one computes the value $f(x)$ for $x = 2.5$ as $f(x) = 5.75$.

A *bilinear interpolation* is an extension of linear interpolation for interpolating functions of two variables $f : \mathbb{R}^2 \to \mathbb{R}$ on a regular grid. The key idea of bilinear interpolation is first to perform linear interpolation in one direction (i.e. x-direction) and afterwards in another direction (i.e. y-direction). In our case we will use bilinear interpolation later to interpolate texture's color values.

a) Show that the linear interpolated value of a function can be computed as following:

$$f(x) \approx f(x_0) + (x - x_0)\frac{f(x_1) - f(x_0)}{x_1 - x_0}, \text{ where } (x \in [x_0, x_1]).$$

b) Derive an equation for computing bilinear interpolated value of $f(x, y)$. You can assume that the values of f at the four points (x_0, y_0), (x_0, y_1), (x_1, y_0) and (x_1, y_1) are given.

Note: $x_0 \leq x \leq x_1$ *and* $y_0 \leq y \leq y_1$

4.3 Texturing (20 Points)

Until now we have only used one color per object. Nevertheless, in reality, e.g. in games, objects are very often colorful because of the usage of textures. This is also a way of making a surface look more geometrically complex. Usually, textures are used by storing *texture coordinates* at the vertices of each triangle and interpolating them to find the correct texel that has to be used for a surface point.

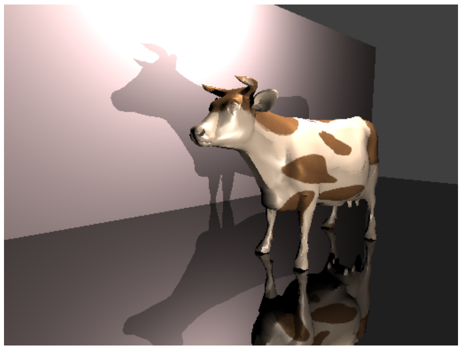
a) In the framework is a new class `TexturedSmoothTriangle` (derived from `SmoothTriangle`), that additionally has the three fields `Vec2f ta,tb,tc`, which correspond to the texture coordinates at vertex a, b, or c, respectively. Add support for texture coordinates to your parser (`ParseOBJ()`).

b) Implement the method `Vec2f TexturedSmoothTriangle::GetUV(Ray &ray)`, which is now a virtual method in your primitive base class. In `TexturedSmoothTriangle`, implement this function to return the x and y coordinates of the interpolated vertex texture coordinates. (For other primitives, just ignore it for now, as we will only use texture-shaders with triangles for now).

c) Implement the `Texture2D::GetTexel()` method to retrieve a color value from the texture image on the given *uv*-coordinates.
Note: clamp the uv-coordinates to the interval $[0 : 1]$.

d) Extend the `PhongShader::Shade()`-method to use the color value given by `Texture::GetTexel()` method sampled on the coordinates given by `Primitive::GetUV()` and combine the texel color with the calculated color by using the vector componentwise product.

If everything is correct your image should look like this:



4.4 Texture wrap modes (Texture tiling) (10 points)

In the previous exercise you have learned how to apply a texture on an object. However if you now apply the *wall.png* texture on the wall object behind the cow, you will encounter results as in the left part of the figure below.
As you may see the applied wall texture isn't mapped intuitively onto the mesh representing the wall. Take a look into the *wall.obj* file, the vertices texture coordinates are not in the range $[0:1]$ anymore. Since we are currently clamping the computed sampling coordinates, see exercise 4.3.c), we can not take advantage of *texture tiling*. In this exercise you have to correct this.

a) Implement the correct behavior of `Texture2D::GetTexel()` to support the `Texture::REPEAT` wrap mode (specified by `Texture::setWrapMode(Texture::REPEAT)`. Instead of clamping texture coordinates you have now correctly map them into the range $[0:1]$, so that a repeating pattern occurs. For example: $1.76 \rightarrow 0.76$, $-2.54 \rightarrow 0.46$.

If everything is correct your image should look like this:

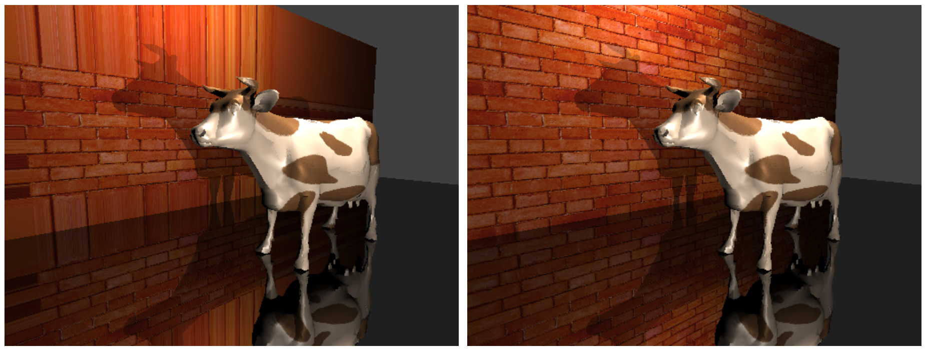


Figure 1: Left: `CLAMP` coordinate wrap mode do clamp the coordinates giving an impression of stretched image boundaries. Right: `REPEAT` coordinate wrap mode produces a repeating pattern of the *wall.png* texture on the wall.

4.5 Texture interpolation (30 Points)

Current implementation does not take into account the possibility of interpolating texture values. We call this interpolation mode *nearest* and specify it by `Texture::setInterpolationMode(Texture::NEAREST)`. This is simple to implement but produces non-smooth results. If you move the camera close to the cow you will encounter the following:

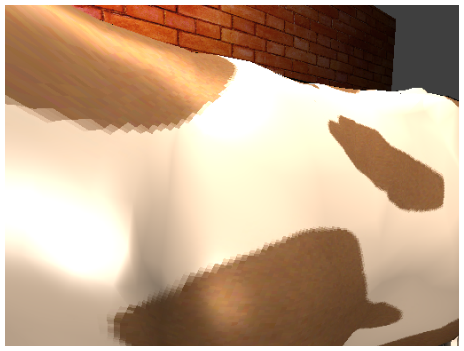


Figure 2: `NEAREST` interpolation mode is fast to compute but produce stair-case patterns.

To overcome this problem one has to interpolate the texture values based on the sampling coordinates. Use the bilinear interpolation equation from the exercise 4.2.b).

a) Implement the correct behavior of the `Texture2D::GetTexel()` method to support `Texture::LINEAR` interpolation mode specified by `Texture::setInterpolationMode(Texture::LINEAR)`. Sample four nearest neighbor texels and perform bilinear interpolation of all color components (i.e. red, green, blue).

Note: The color values of the texels has to be defined in the center of the pixel. Thus the neighboring texels are the four neighbors surrounding the sampling position. This mimics the OpenGL's behavior on handling with texture samplings. The following image illustrates it:

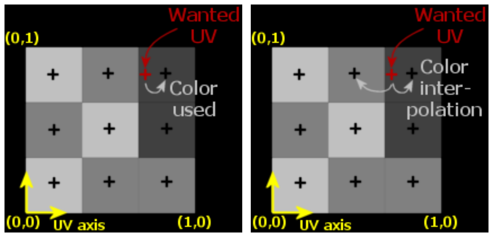
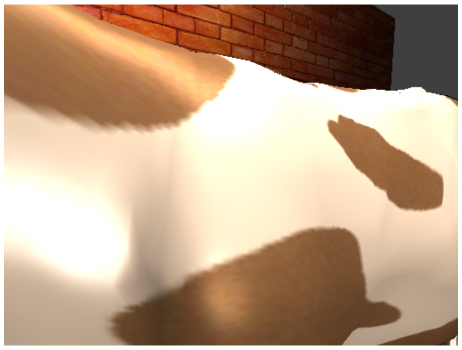


Figure 3: OpenGL's texel sampling interpolation modes. Left: nearest interpolation samples the nearest texel. Right: linear interpolation interpolates between four neighbors (in this image only left and right neighbors are used)

If you have implemented the interpolation correct your results should look like this:



There is a small debug scene defined in the `MicroTracer.cxx` which renders a quad with applied *debug.png* texture on it. You can test your interpolation implementation within this scene. If your interpolation is correct you should achieve following results:

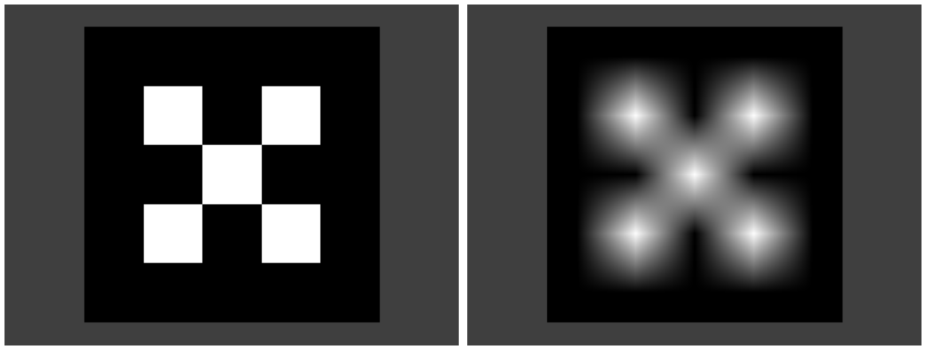


Figure 4: Left: Nearest interpolation mode. Right: linear interpolation mode. Note that such a star like pattern occurs when you place the texel value in the center of a pixel.

4.6 Analytical solution of the rendering equation in 2D (30 Points)*

This exercise is voluntary and can be handed in for bonus-points.
The Figure 5 shows a simple 2D scene with a linear light source L of uniform radiance 1 for each point and direction. Assume that the light source absorbs all light hitting it. Located at the $y = 0$ line is a Lambertian material with the following BRDF:

$$f_r(\omega, (p, 0), \omega_o) = \frac{1}{\pi}$$

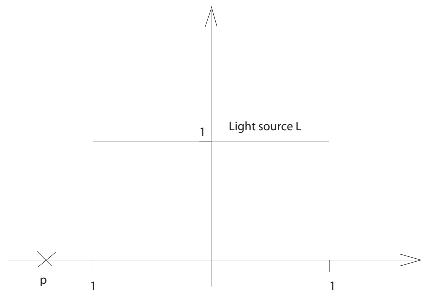


Figure 5: The linear light source reaches from -1 to 1 at y-position 1 with a uniform radiance of 1 for each point on the light source and each direction.

a) The standard rendering equation in 2D is given by:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_0^{\pi} f_r(\omega, x, \omega_o) \cdot L(x, \omega) \cdot cos\phi \cdot d\omega$$

Solve the rendering equation analytically for each point $x = (p, 0)$ and direction ω_o. As the rendering equation shows, you have to integrate over the hemicircle for each point x.

b) Let S be the set of all points on the light source (the $y = 0$ plane can be ignored here) then the point form of the rendering equation in 2D is given by:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{y \in S} f_r(\omega_i, x, \omega_o) \cdot L(y, -\omega_i(x, y)) \cdot \frac{cos\phi_i cos\phi_y}{|x - y|} \cdot dA_y$$

Note that the denominator $|x - y|$ is really the distance not the squared distance as in the 3D version. Again solve the equation analytically, but now by integrating over the light source.

Hint: There is a Maple installation in the cip-pools that can be used to solve, especially the second part of the exercise. Type `/usr/local/maple8/bin/xmaple` to start Maple. Integration is performed by typing

```
int(sin(Pi*x),x=-2 ...  1);
```

for instance to get the integral of the function $sin(\pi x)$ with ranges -2 to 1.