UNIVERSITÄT DES SAARLANDES
PROF DR. PHILIPP SLUSALLEK
LEHRSTUHL FÜR COMPUTERGRAPHIK
STEFAN POPOV (POPOV@GRAPHICS.CS.UNI-SB.DE)
SVEN WOOP (WOOP@GRAPHICS.CS.UNI-SB.DE)

20TH NOVEMBER 2006

COMPUTER GRAPHICS I
ASSIGNMENT 4

Submission deadline for the exercises: Thursday, 30th November 2006

4.1 Vertex Normals (20 Points)

Rather than storing a single *geometry normal* for a triangle, it is often useful to store at each vertex a corresponding *vertex normal*. The advantage is that if we have a hit point on a triangle, the shading normal can be smoothly interpolated between the vertex normals. If neighboring triangles share the same vertex normals, a smooth appearance can be generated over non-smoothly tessellated geometry.
Proceed as follows:

a) Download the new microTrace framework from
http://graphics.cs.uni-sb.de/Courses/ws0607/cg/microTrace04.zip.

b) Extend `Scene::ParseOBJ` to also support vertex normals. Take a look at the included `.obj` files.

c) Turn off BVH-support.

d) Your ray class is extended with two additional `float` values `Ray::u` and `Ray::v`.

e) In `Triangle::Intersect`, store the computed barycentric coordinates into `Ray::u` and `Ray::v`.

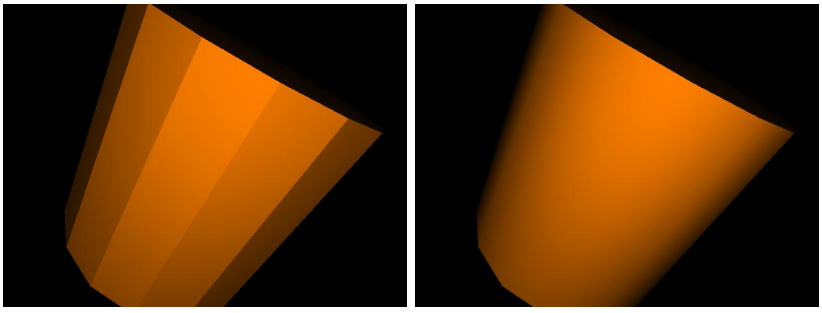
Note: As long as your other classes (e.g. `Sphere`) don't need local surface coordinates, there is no need to compute them yet.

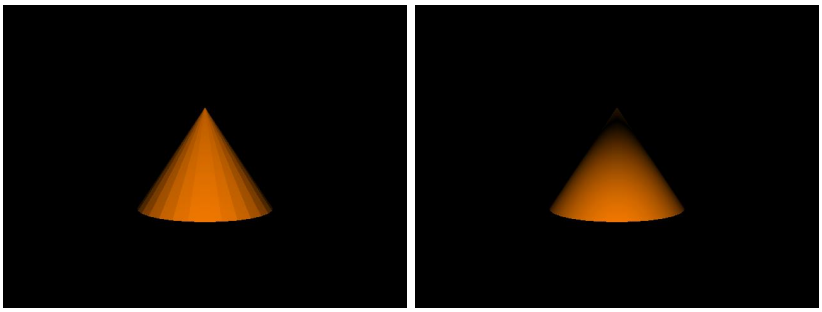
f) In the framework is a new class `SmoothTriangle` which stores the vertex normals (`na, nb` and `nc`) additionally to the original vertex positions.

g) In `SmoothTriangle::GetNormal()` use the u/v coordinates of the hit-point to interpolate between the vertex normals. Note: Interpolating normalized vectors will not return a normalized vector ! Make sure to normalize your interpolated normal !

h) Test your implementation with *cylinder16.obj cone32.obj* using the appropriate camera you can choose in `Scene.hxx`. Compare the difference between the regular and the smooth triangles.

If everything is correct your images should look like this:





4.2 Procedural Bump Mapping (20 Points)

In the last exercise you have learned that the appearance of a surface can be changed by using a modified *surface normal*. In this exercise we will implement a technique called *bump mapping*, which modifies the surface normal such that a surface gives the impression of being *bumpy*. This allows to generate the appearance of highly complex surface with only very few primitives. In order to do this, three parameters have to be known for each surface point:

- The original surface normal N.
- A local coordinate frame for this surface point. Even any coordinate frame can be used (as long as it is consistent between closeby points), the usual way is to use the surface derivates in u and v direction, called $dPdu$ and $dPdv$.
- The values Δ_u, Δ_v stand for the amount of change along these tangent vectors. This amount is usually either read from a texture or is computed procedurally. The final normal during shading (also for reflections) is then $N' = Normalized(N + \Delta_u dPdu + \Delta_v dPdv)$.

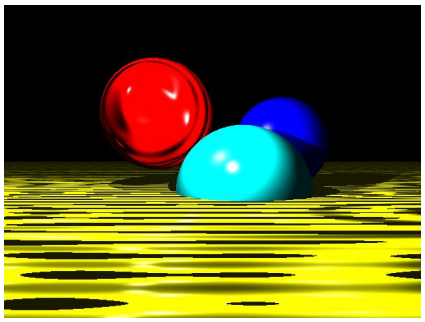
In this exercise, you will implement a very basic version of procedural bump mapping:

- As surface derivatives, use $dPdu = (1, 0, 0)$ and $dPdv = (0, 0, 1)$.
- For the amount of deviation, use a simple procedural approach, by computing $\Delta_u = \frac{1}{2}\cos(3H_x \sin H_z)$, $\Delta_v = \frac{1}{2}\sin(13 * H_z)$. H denotes the hit point of the ray with the surface.

For your implementation, proceed as follows:

a) Implement the `Shade`-method in `BumpMappedPhongShader.hxx` by first copying the `Shade`-method from the basic Phong shader and then modifying the normal at the beginning of the `Shade` function, following the guidelines given above.

If your shader works correct you should get an image like this using the scene description in `MicroTrace.cxx`:



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.

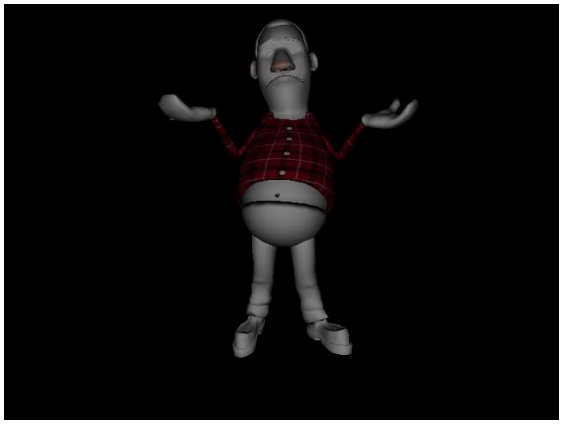
a) Turn BVH-support on

b) In the framework is a new class `TexturedSmoothTriangle` (derived from `SmoothTriangle`), that additionally has the three fields `Vec3f ta,tb,tc`, which correspond to the texture coordinates at vertex a, b, or c, respectively. For the sake of simplicity we will use `Vec3f`'s to store the texture coordinates, even though they usually require only 2 coordinates (barycentric coordinates). Add support for texture coordinates to your parser (`ParseOBJ()`).

c) Implement the method `Vec3f TexturedSmoothTriangle::GetUV(Ray &ray, float &u, float &v)`, 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 will only use texture-shaders with triangles for now).

d) Implement the `TexturedEyeLightShader::Shade`-method to use the texture coordinates returned by `GetUV` and combine the texel color with the calculated eye-light color using the vector product.

Test your implementation on `barney.obj` with the texture `barney.ppm`. If everything is correct your image should look like this:



4.4 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 *super-sampling*, i.e. to shoot more than one ray per pixel. The three most frequently used super-sampling 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`.
- 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 `GetSamples`-method in `RegularSampleGenerator.hxx`, `RandomSampleGenerator.hxx`, and `StratifiedSampleGenerator.hxx`, which are derived classes from `SampleGenerator`.

Use `ground.obj` and `cb.ppm` to render your image with 4 samples and compare them to the images you can download:
http://graphics.cs.uni-sb.de/Courses/ws0607/cg/regular.jpg.
http://graphics.cs.uni-sb.de/Courses/ws0607/cg/random.jpg.
http://graphics.cs.uni-sb.de/Courses/ws0607/cg/stratified.jpg.