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)

12. November 2007

Computer Graphics I
Assignment 3

Submission deadline for the exercises: Monday, 19th November 2007

3.1 Reflection Rays (5 Points)

Given a ray $R(t) = O+t\cdot D$ which hits a reflective surface at $t = t_{hit}$. The surface has the geometry normal N at the hit point. Assume that both, the ray direction D and the surface normal N are normalized.

a) Compute the ray that has been reflected (assuming a perfect mirror reflection) by the surface. You have to submit the solution for this exercise in written form.

3.2 Front/Back-side of a triangle (5 Points)

A triangle has two sides, front-side and back-side, in a 3D space. The normal vector N points away from the front side. Given a ray $R(t) = O + t \cdot D$ which hits a triangle and a face normal N of the hit point.

a) Find a way of how to determine if the ray hits the face from the front or from the back. You have to submit the solution for this exercise in written form.

3.3 New classes and changed application logic (5 Points)

Download the new `microTrace`-framework from
http://www.mpi-inf.mpg.de/departments/d4/teaching/ws2007-2008/cg/microTrace03.zip
Study the new framework-code of `Shader.hxx`, `Scene.hxx`, `Light.hxx`, `MicroTrace.cxx`. There is a new class `Scene` which handles the complete scene environment (i.e. objects and lights).

a) Implement the `Scene::Intersect(Ray& ray)` method. The method perfroms in the same way as its counterpart in the `Object` class. However instead of intersecting primitives the nearest intersection with the scene objects has to be found.

3.4 The surface shader concept (10 + 10 Points)

A *surface-shader* is a small program that is assigned to a primitive and is responsible for computing the color of each ray hitting this primitive. For example, a *flat shader* might just return a constant color for a primitive, whereas another shader might compute more complex effects such as lighting, shadows, or texturing.

Each primitive has a pointer `Shader *shader` in the new framework. The `Object` class has a method `Object::setShader(Shader*)`. Implement this method in such a way, that applying a shader to the object will apply it to every primitive handled in it. For example, our cow (see assignment 2) could be initialized with `cow->setShader(new DebugShader(scene));`. This should apply the shader `DebugShader` to every primitive in the cow model. As we will see later some shaders need access to the scene data (e.g for light or shadow calculations), this is why each shader gets a pointer to the scene objects.

In this exercise, you will add some missing parts of the given basic shader framework to your ray tracer and implement two simple shaders:

a) Implement a simple debug shader, which uses the primitive's normal to colorify the ray. (see assignment 2). Proceed as follows:

- The shader class has a purely virtual function `Vec3f Shader::Shade(Ray &ray)`, which has to be implemented in all derived shaders.
- Implement the `DebugShader::Shade(Ray& ray)` method. The method should just return the absolute value of the normal vector (x, y, z) as color values (r, g, b).

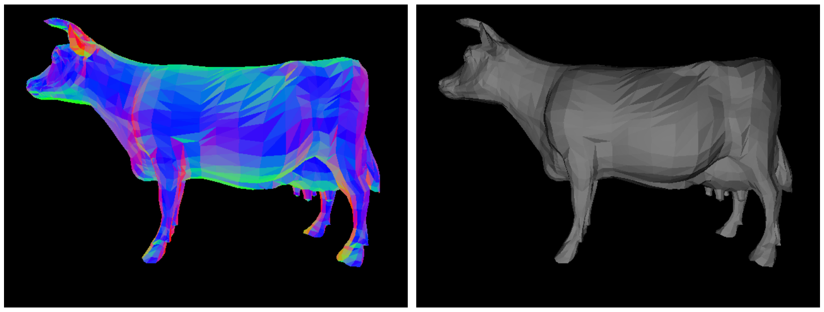
b) Implement the `Shade`-method in the *eye light* shader, which uses the angle between the incoming ray and the surface normal at the hit point to achieve a better impression of the actual primitive's shape. The resulting color should be calculated according to:

$$result = |\cos\theta| \cdot color$$

where θ is the angle between the primitive surface normal and the ray direction. The shader needs to know some information about the primitive (i.e. the surface normal).

- Implement the shading function `EyeLightShader::Shade(Ray &ray)` using `ray.hit->GetNormal(ray)` to retrieve the surface normal of the primitive. With the surface normal the above given formula can be applied.

If the test scene specified in `MicroTrace.cxx` is rendered with these two shaders (i.e. `DebugShader` and `EyeLightShader`) the resulting images (`result_3.4a.ppm` and `result_3.4b.ppm`) should look like this:



3.5 Vertex Normals (15 Points)

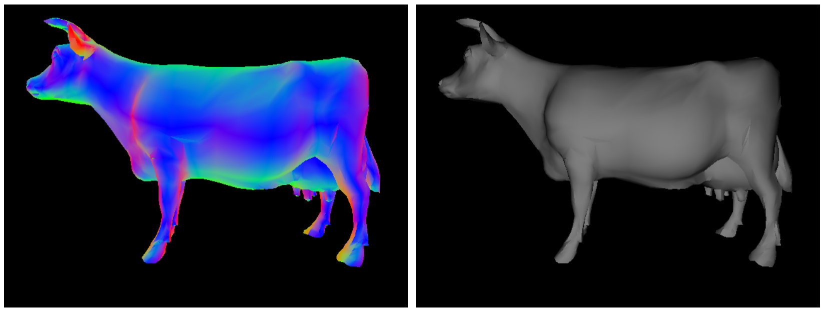
Rather than storing a single *geometry normal* for a triangle, it is often useful to store at each vertex a corresponding *vertex normal*. The advantage is that if we have a hit point on a triangle, the shading normal can be smoothly interpolated between the vertex normals. If neighboring triangles share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

a) Extend `Scene::ParseOBJ` to also support vertex normals. Take a look at the included `.obj` files. (5 Points)

b) Your ray class is extended with two additional `float` values `Ray::u` and `Ray::v`. In `Triangle::Intersect`, store the computed barycentric coordinates into `Ray::u` and `Ray::v`.
Note: From now on we will handle only triangles as primitives. Hence As long as your other classes (e.g. `Sphere`) don't need local surface coordinates, there is no need to compute them yet.

c) In the framework is a new class `SmoothTriangle` which stores the vertex normals (`na, nb` and `nc`) additionally to the original vertex positions. In `SmoothTriangle::GetNormal()` use the u/v coordinates of the hit-point to interpolate between the vertex normals. (10 Points)
Note: Interpolating normalized vectors will not return a normalized vector ! Make sure to normalize your interpolated normal!

If everything is correct your images (i.e. `result_3.5a.ppm` and `result_3.5b.ppm` should look like this:



Compare the difference between the regular (exercise 3.4) and the smooth triangles.

3.6 Mirror shader(10 Points)

In this exercise you have to implement a mirror shader. Each ray which hits a surface with applied mirror shader has to be reflected according to the exercise 3.1. The reflected ray start the traversal of the scene from the point of reflection.

a) Implement the missing parts of `MirrorShader.hxx`. (10 Points)

b) Add new object *plane.obj* to your scene which already contains the cow model. Apply the mirror shader to the new object.

If your shader is implemented correct you should achieve following results:

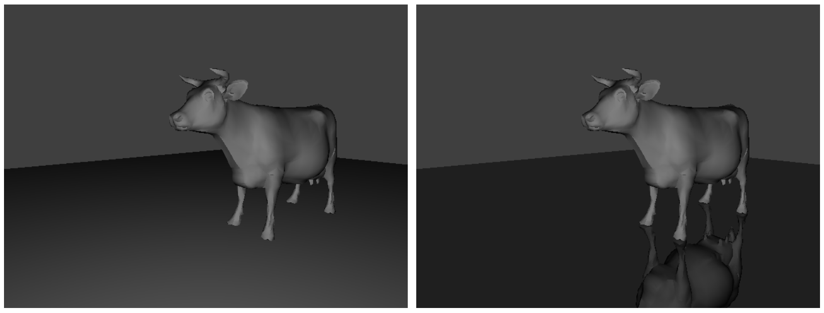


Figure 1: Plane with applied shader: Left `EyeLightShader`, Right `MirrorShader`

3.7 Phong Shading and Point Light sources (30 Points)

In the last exercise we implemented two simple surface shaders, which do not take light sources into account. A more advanced surface shading concept, the *Phong shading model*, utilizes light sources to increase the rendering realism and give objects a plastic like appearance. Before we can implement the `Shade` method in `PhongShader.hxx` we have to implement a simple light source.

a) Implement a point light. Proceed as follows: (10 Points)

- Study the base class `Light.hxx`. Each light source which we will derive from it has to implement an `Illuminate(Ray &ray, Vec3f &intensity)` method.
- Implement the `Illuminate`-method of `PointLight.hxx`. The method should calculate the light intensity based on the quadratic falloff (i.e. $\frac{1}{t^2}$, where t is the distance between light source and surface point), as well as the direction vector from the surface point to the light source. The direction vector will be later used for shadow computations.

b) Implement the *Phong illumination model* (20 Points)

- The value L_r returned by `PhongShader::Shade()` should be calculated according to:

$$L_r = k_a c_a L_a + k_d c_d \sum_{l=0}^{n-1} L_l(\mathbf{I_l} \cdot \mathbf{N}) + k_s c_s \sum_{l=0}^{n-1} L_l(\mathbf{I_l} \cdot \mathbf{R})^{k_e}$$

  c_a: Ambient color
  c_d: Diffuse color
  c_s: Specular color (use $c_s = (1, 1, 1)$)

  k_a: Ambient coefficient
  k_d: Diffuse coefficient
  k_s: Specular coefficient
  k_e: Exponent (*shine* parameter)

  L_a: Ambient radiance (use $L_a = (1, 1, 1)$)
  L_l: Radiance arriving from light source l

  $\mathbf{I_l}$: Direction to light source l
  $\mathbf{N}$: Shading normal
  $\mathbf{R}$: Reflected incident ray direction (points away from the surface)

  n: Number of light sources

- Add to your existing scene a point light with following parameters: `position=Vec3f(0.1, 0.03, -0.2)` and `intensity=Vec3f(0.05, 0.05, 0.05)`.
- Attach the phong shader to the cow model. The phong shader parameters has to be set up as following: $c_a = c_d = (0.8, 0.7, 0.6)$, $k_a = 0$, $k_d = 0.8$, $k_s = 0.4$, $k_e = 12.0$.

If your shader is implemented correctly you should achieve following results:

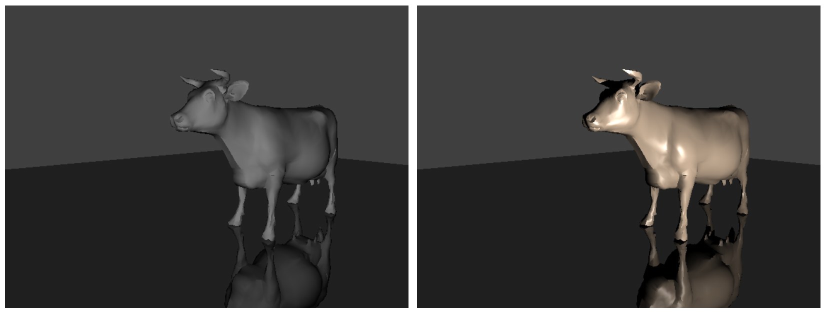


Figure 2: Left: cow with `EyeLightShader`, Right: cow with `PhongShader`

Notes:

- Sometimes an incident ray may hit the backside of a surface. Then, just turn the shading normal around to face forward.
- Only consider light sources that illuminate the primitive from its front-side.

3.8 Shadows (10 Points)

To add more realism to the Phong model we want now to incorporate shadows into it. Proceed as follows:

- Modify `PhongShader::Shade` to check for occlusion. Incorporate the distance returned by `Scene::Intersect()` method, which you have implemented in exercise 3.3. (10 Points)
- Add a new object into the scene (i.e. *wall.obj*) and apply phong shader on it. The parameters of the wall's phong shader are: $c_a = c_d = (0.8, 0.7, 0.6)$, $k_a = 0$, $k_d = 1.0$, $k_s = 0.0$, $k_e = 64.0$.

If everything is implemented correctly your images should look like in Figure 3.

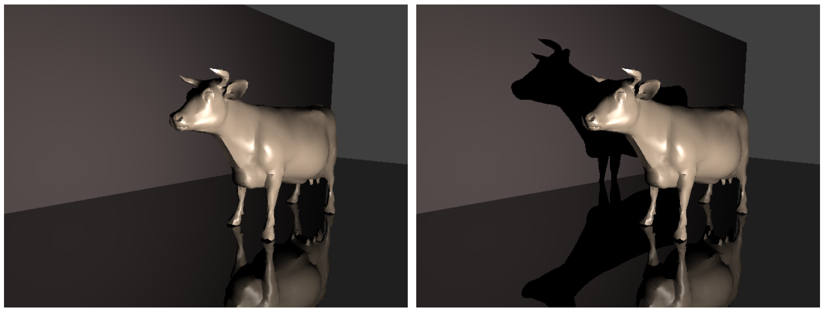


Figure 3: Left: without shadows, Right with shadows

3.9 Area Lights* (20 Points)

As you have learned in the last exercise, shadows can add important visual information to an image. Until now we have only considered point lights. Point lights create *hard shadows* because a point light can not be partly occluded and is either blocked or not. To render more realistic shadows we need a more advanced light source. *Area Lights* are able to produce *soft shadows*, which are more natural. In this exercise we implement a `QuadAreaLight` (in `QuadAreaLight.hxx`) which is defined by a position p and spanning vectors e_1 and e_2.

- Calculate the normal and the area of the QuadAreaLight in the constructor (5 Points)
- In each illumination step do place a shadow ray randomly along the light surface area (use `frand()`, see `stdafx.cxx`). Compute the illumination similar to the point light but incorporate the area value into your calculations. (15 Points)
- Add a quad area light to the scene with following parameters: `intensity = Vec3f(2.0, 2.0, 2.0)` $p = (0.1, 0.03, -0.2)$, $e_1 = (0.15, 0.0, 0.0)$, $e_2 = (0.0, 0.15, 0.0)$ and remove the point light.
- Render an image with 1000 shadow rays per pixel (drink a coffee, since this takes time ;-))

If everything is implemented correctly your images should look like this (left 10 shadow rays, right 1000):