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)

06. November 2006

Computer Graphics I
Assignment 2

Submission deadline for the exercises: Thursday, 16. November 2006, before the lecture

2.1 Reflection Rays (5 Points)

a) 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. 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.

2.2 Encapsulate camera and primitives from main application logic (5 Points)

a) Download the new `microTrace`-framework from
http://graphics.cs.uni-sb.de/Courses/ws0607/cg/microTrace02.zip
(Available at Thursday, 9th November)

b) Study the new framework-code of `Shader.hxx`, `FlatShader.hxx`, `EyeLightShader.hxx`, `Scene.hxx`, and `microTrace.cxx`

c) A pointer `Primitive *hit` is now contained in your `Ray` class. After a ray has been successfully intersected with a primitive, store the primitive's address in `hit` (if the hit ditance is smaller than `ray.t`).

d) In the class `Scene` you find a method `Add(Primitive*)`. Change your code accordingly using the appropriate `vector` defined in the class.

e) Rather than intersecting each primitive in the main function, we will now use the `Intersect(Ray &ray)` method of the `Scene` class. After modification the method should iterate over all primitives, intersect them and return `true` or `false` depending on if we had a valid hit with the scene data or not.

f) The loop of `microTrace.cxx` calls the `RayTrace(Ray &ray)` method of `Scene`. The `RayTrace` method should call `Intersect` and depending on a hit or none return a white or black color.

2.3 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.

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 flat shader. Proceed as follows:

- The shader class has a pure virtual function `Vec3f Shader::Shade(Ray &ray)` , which has to be implemented in all derived shaders.
- Implement the `FlatShader::Shade(Ray &ray)` method. The method should just return the color passed in the constructor of `FlatShader`.
- Each primitive has a pointer `Shader *shader` in the new framework and a corresponding modified `Constructor` definition. Adjust the `Constructor` code appropriate. For example, our red sphere could be initialized using `Sphere s1(Vec3f(-2,1.7,0),2,new FlatShader(Vec3f(1,0,0), &scene));`. 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.
- Finally, if, for instance, the primitive intersected by a ray has been stored in `Primitive *hit`, the appropriate color can then be computed by calling `hit->shader->Shade(ray);` Change your code in `RayTrace` such that not black or white is returned but the color from the primitive with the closest hit or the background color if a ray does not hit a primitive.

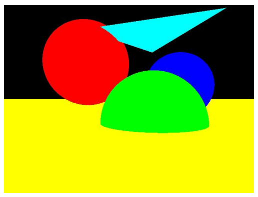
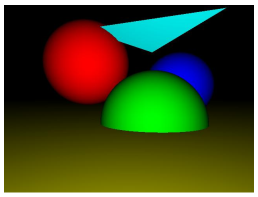
b) Implement the `Shade`-method in the *eye light* shader, which uses the angle between the incoming ray and the surface normal at the hit point to achieve a better impression of the actual primitive's shape. The resulting color should be calculated according to:

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

where θ is the angle between the primitive surface normal and the ray direction. As the shader now needs to know some information about the primitive (i.e. the surface normal), some modifications are necessary:

- Implement the `Vec3f Primitive::GetNormal(Ray &ray)` method in all classes derived from `Primitive`. `GetNormal(Ray &ray)` should return the **normalized** normal of the primitive. The `ray` parameter passed to `GetNormal(Ray &ray)` should be a ray that has been successfully intersected before, so you can assume that the intersection stored in this ray corresponds to the actual primitive. For example, `ray.org+ray.t*ray.dir` should be a point on the primitive.
- 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 it should look like:

2.4 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:

- 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 `Add(Light*)`-method of the `Scene`-class.
- Implement the `Illuminate`-method of `PointLight.hxx`. The method should calculate the light intensity, as described in the lecture, which hits the surface point from the light source 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*

- 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 lights sources

Notes:

- Sometimes an incident ray may hit the backside of a surface (i.e. the shading normal points to the other side.) Then, just turn the shading normal around to face forward.
- Only consider light sources that illuminate the primitive from its front-side (i.e. $\mathbf{I_l} \cdot \mathbf{N} > 0$).

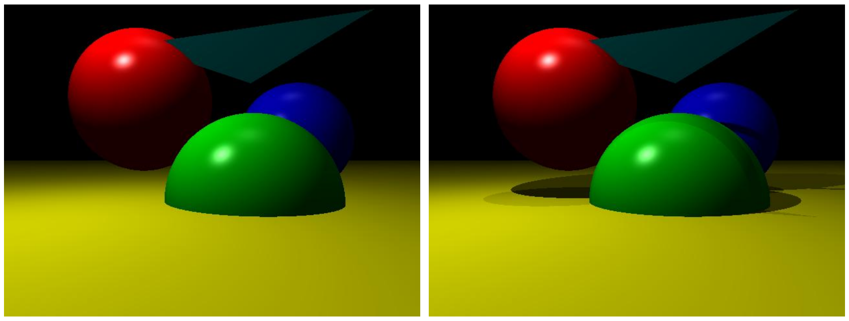


Figure 1: Left: without shadows, Right with shadows

2.5 Shadows (20 Points)

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

- Implement the method `Occluded` in the `Scene`-class, which should check if something blocks the light.
- Modify `PhongShader::Shade` to check for occlusion.

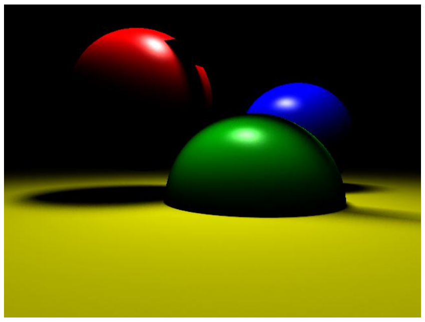
If everything is implemented correct your images should look like in Figure 1.

2.6 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 four points in space:

- Calculate the normal and the area of the QuadAreaLight in the constructor
- Calculate the intensity as described in the lecture by generating a random sample position on the area light (using `drand48()` and bi-linear interpolation).
- Add `QuadAreaLight quadLight(&scene, quadLightIntensity, Vec3f(-1.5, 10, -1.5), Vec3f(1.5, 10, 1.5), Vec3f(1.5, 10, -1.5), Vec3f(-1.5, 10, 1.5));` to `microTrace` and remove the point lights.
- Render an image with 1000 shadow rays per pixel

If everything is implemented correct your images should look like this:



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

This exercise is voluntary and can be handed in for bonus-points.
The Figure 2 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:

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

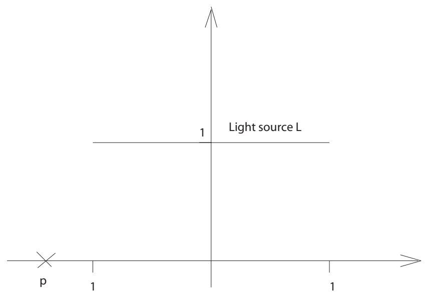


Figure 2: The linear light source reaches from -1 to 1 at y-position 1 with a uniform radiance of 1 for each point on the light source and each direction.

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.