Introduction to Computer Graphics
Exercise 2 – Raytracing

Handout date: 04.10.2012
Submission deadline: 17.10.2012, 14:00 h

Note

Undeclared copying of code or images (either from other students or from external sources) is strictly prohibited! Any violation of this rule will lead to expulsion from the class. Late submissions are not accepted.

What to hand in

A .zip compressed file named "Exercise*n*-Name1-Name2.zip" where *n* is the number of the current exercise sheet and the names are the full names of the submitting students. It should contain:

- A zipped folder containing a Microsoft Visual Studio 2005 file solution with all source files and project files. In order to avoid sending build files, close your Visual Studio solution and run the file "remove-compiled.bat" located in the top directory of the framework before you submit. Unzip and rebuild your submission to make sure it compiles without errors.
- A "readme.pdf" file describing your approach to the solution (1-2 sentences per exercise) and any problems you encountered(use the same numbers and titles).
- Submit your solutions to the class moodle before the submission deadline. If you work in a pair, it is enough if one of you submits it, just put both names in the title of the zip file.

"Rays Never Stop, or do they?"

The goal of this exercise is to implement a more sophisticated shading model and extend the RayTracer to enable recursive effects such as reflection, refraction, and shadows.
Those of you who want to use their own solutions from Exercise 1 we ask to copy your code into the new framework provided with exercise 2.

2.1 Shading

2.1.1 Normals and Materials (6 Points)

In order to shade the objects more elaboratly than with constant colors, the shaders need more information about the intersection point. In the `intersect` routine of every geometric element (Plane and Triangle), add the `normal` and `material` at the intersection point to `iData` in case of a valid intersection. The `PhongLightingShader` will also need the origin of the ray that intersected the object (`sourcePosition`). Use the corresponding fields of the class `IntersectionData` to store these values.
Note:

- The triangle has one normal for each vertex. Use these normals to interpolate the normal at the intersection point using barycentric coordinates.

- This task is not mandatory for any additional shapes you implemented, e.g. the `Quadric` from exercise 1, but it will make your renderings look cooler.

2.1.2 Diffuse Lighting Model (6 Points)

A diffuse lighting shader follows the diffuse lighting model and uses the surface normal and (only) the diffuse component of the material at the intersection point to compute the shaded color for every light source in the scene. The intensity I resulting from one light source is given by $I_p \odot k_d(\mathbf{N} \cdot \mathbf{L})$, where $\odot$ denotes element-wise multiplication. I_p is the color of the light source, k_d the diffuse material color, $\mathbf{N}$ the normal at the intersection and $\mathbf{L}$ the normalized vector from the intersection towards the light.
Implement this lighting model in the method `DiffuseLightingShader::shade`. The colors resulting from different light sources are added up and clamped to [0,1] at the end. To use the diffuse lighting shader, change the renderer configuration in the file *Config.xml*. Hints: If the light ray and normal point in opposite directions, the current light doesn't contribute to diffuse shading.
Render the scene *SceneDescription_ex2.xml*, compare it to figure 1, and provide your rendering in your *readme.pdf*.

2.1.3 Phong Lighting Model (12 Points)

A Phong lighting shader follows the Phong lighting model and uses the normal, material and viewpoint to compute the shaded color for every light source in the scene. Implement this lighting model in the method `PhongLightingShader::shade`. The intensity I resulting from one light source is given by $I_a \odot k_a + I_p \odot (k_d(\mathbf{N} \cdot \mathbf{L}) + k_s(\mathbf{V} \cdot \mathbf{R})^n)$. I_a is the ambient light intensity, k_a the ambient reflection color of the scene, k_s the specular color of the material, $\mathbf{V}$ the normalized vector from the intersection towards the camera (view vector), $\mathbf{R}$ vector $\mathbf{L}$ reflected along the normal and n the shininess of the material. The colors resulting from different light sources are added up and clamped to [0,1] in the end. **Use** the `sourcePosition` of the intersection data to compute the viewing direction. To use the Phong lighting shader, change the renderer configuration in the file *Config.xml*.
Hint: If the light ray and normal point in opposite directions, the current light doesn't contribute to the diffuse and specular component. If the viewing ray and the reflected light ray point in opposite directions, then the current light does not contribute to the specular component.
Render the scene *SceneDescription_ex2.xml*, compare it to figure 1, and provide your rendering in your *readme.pdf*.

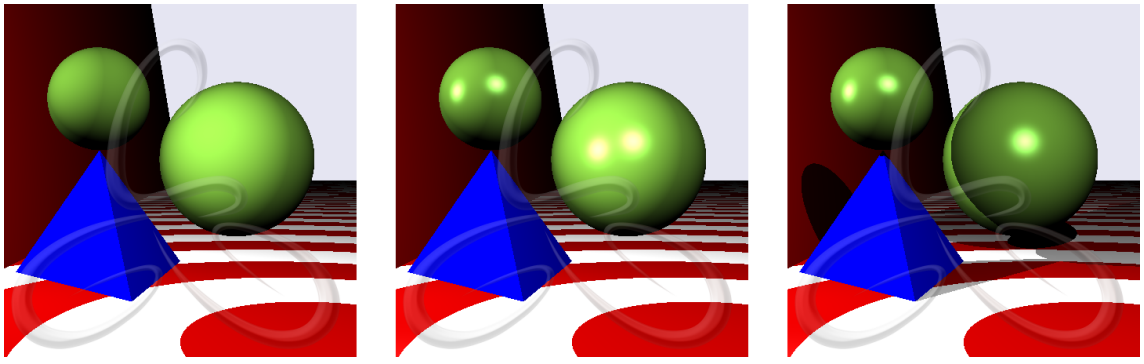

Figure 1: Diffuse lighting, phong lighting and shadows in *SceneDescription_ex2.xml*. *left:* Ex 2.1.2, *middle:* Ex 2.1.3., *right:* Ex 2.2.1. Watermarked.

2.2 Ray Tracing

2.2.1 Shadows (6 Points)

An intersection point is in shadow if the light source is occluded by other geometry. Implement the method `Scene::getNonOccludedLights` using the method `Scene::fastIntersect` to get a list of lights visible from a given point. Use the method `ILight::generateRay` to generate a ray from a light source going through a point. The method `Scene::fastIntersect` test if a ray intersects with the scene. Update the methods `DiffuseLightingShader::shade` and `PhongLightingShader::shade` to use only non-occluded light sources for their calculations. Render the scene *SceneDescription_ex2.xml*, compare it to figure 1, and provide your rendering in your *readme.pdf*. Explain why `PointLight::generateRay` sets `max_t` of the ray.

2.2.2 Additional Intersection Information for Recursion (1 Point)

For the remaining tasks, additional data will have to be stored for each intersection point. In the `intersect` routine of every geometric element, add the `reflectionPercentage` of the element to `iData` in case of a valid intersection. Use the corresponding fields of the class `IntersectionData` to store these values.

2.2.3 Reflection (12 Points)

Extend the method `Renderer::traceColor` to recursively compute the shading color. `ReflectionPercentage` is a value between 0 and 1 and determines the percentages of the final color computation that stem from reflected rays. The final color at a point is given by $\texttt{reflectionPercentage} * reflectedShading + (1 - \texttt{reflectionPercentage}) * objectShading$. Use the intersection point's normal to generate a reflection ray. Hint: Increase the argument `recDepth` by one at each recursive call. The recursion depth is defined in *Config.xml* which loads into `Renderer::m_recursionDepth`. The ray's constant value `epsilon_t` plays an important role to avoid self intersections due to numerical artifacts. Use it to limit the ray's `min_t`. What happens if you do not use this offset?
Render the scene *InfiniteRoom.xml* with recursion depths (0, 1, 2 and 8). Compare the results to figure 2, and provide your renderings in your *readme.pdf*.

2.2.4 Refraction* (For the passionate)

Refraction models translucent objects by allowing rays to pass through the objects. First update the `intersect` routines to add the `refractionPercentage` and `refractionIndex` of the element to `iData` in case of a valid intersection. For the refraction computation you will also need information on whether the ray enters or leaves the object at the intersection point. Thus compute the boolean `rayEntersObject` for every intersection. `RefractionPercentage` is a value between 0 and 1 and determines the percentages of the final color computation that stem from refracted rays. You can compute the final color at a point as

$$P_{refl} * S_{refl} + P_{refr} * S_{refr} + (1 - P_{refl} - P_{refr}) * S_{obj}$$

where P are percentages and S are shadings. Extend the method `Renderer::traceColor` to compute refraction rays according to Snell's Law. Note that the scene stores its own refraction index (index of refraction for the scene's medium, e.g. air), which can be set in the scene description file.

2.3 Final Rendering

2.3.1 Render a Bigger Mesh (2 Points)

Render the scene *Cornell.xml*, which contains a mesh with 588 triangles (loaded into `MeshTriangle`). Edit the renderer configurations in the file *Config.xml* for Phong lighting model and recursion depth of 8. Compare the result to figure 3, and provide your rendering in your *readme.pdf*. Also

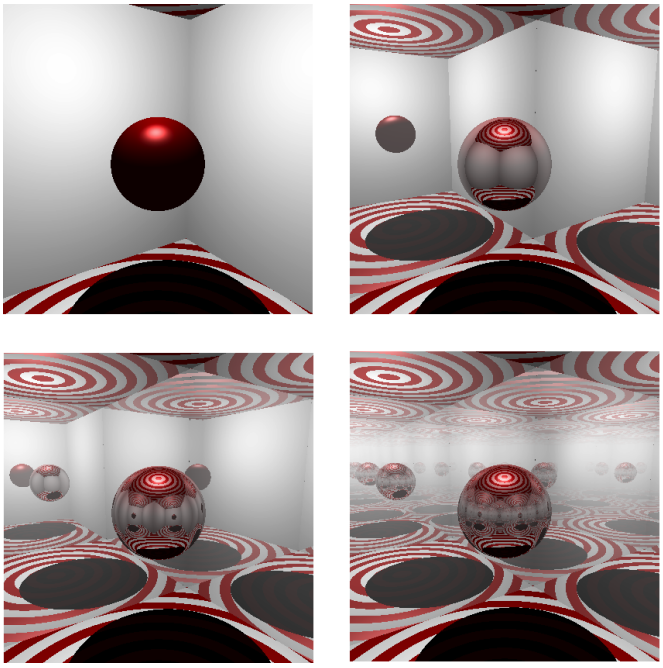

Figure 2: Phong lighting of *SceneDescription_infiniteRoom.xml* with different recursion depths. *Top-left:* 0, *Top-right:* 1, *Bottom-left:* 2, *Bottom-right:* 8.

write down the duration of the rendering as reported in the console. Use Release Mode, and be patient, it can take a while.

2.4 More Geometry

2.4.1 Torus* (For the passionate)

Torus-ray intersections are nontrivial. Implement the `intersect` routine for the `Torus` similarly to the ones of the last exercise sheet. (Hint: The routine `rpoly.h` in the *utils* folder might come in handy for this task.)

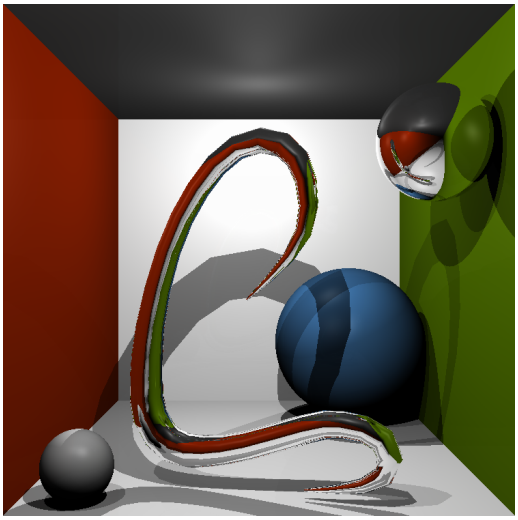

Figure 3: The final rendering of Cornell.xml.