Computer Graphics DT 3025
Lab 4

Stevan Tomic and Martin Magnusson, November 2015
Based on material by Mathias Broxvall

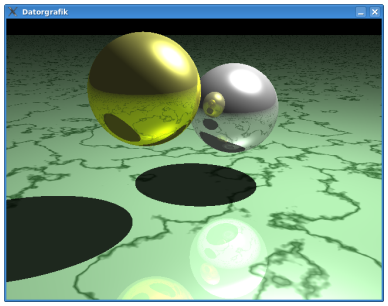

Figure 1: A simple realtime raytracer with procedural textures.

Lab 4 - Raytracing

With the ever growing speed, and most importantly number of, processors in modern computers it is becoming more and more common to use raytracing both for offline and for online generation of images.

Some of the advantages of using raytracers is the simplicity with which advanced features such as true reflections, transparent materials with refractions, procedural textures, shadows and other features can be implemented. The major disadvantages, so far, is the large computational cost which still limits the use of raytracing significantly.

In this exercise you will test an alternative method of generating graphics, compared to OpenGL rasterised graphics in the previous labs, by using and extending a small realtime raytracer.

Start by downloading and compiling the files in `lab-4.zip`. To properly compile this program under Microsoft Visual Studio you still need to include and link with SDL as in the previous labs, because the code uses SDL for the window managment, but OpenGL and GLM etc. are not used in this lab.

In order to use all available processor cores on the machines, you need to also enable *OpenMP* which is a mechanism built in to many compilers in order to utilize multiple threads for computations. To do this you need the following settings under *project properties*:

- Configuration: `Active(release)`. This is important for optimizing the speed, since otherwise the program will run at only a tenth of the proper speed.
- Linker → Input → Additional Dependencies: `SDL.lib SDLmain.lib`
- C/C++ → Optimization: `Maximize speed (/O2)`
- C/C++ → Language → OpenMP Support: `Yes (/openmp)`

When you now compile and run the program you will see a simple scene containing two reflective spheres bouncing on a glossy marbled floor, see Figure 1. This scene is raytraced in realtime and will use as many processors as are available in the computer and can achive approximately 90 FPS on the CPUs in the computer lab. By holding down the left mouse button and dragging the mouse over the window you can rotate the camera.

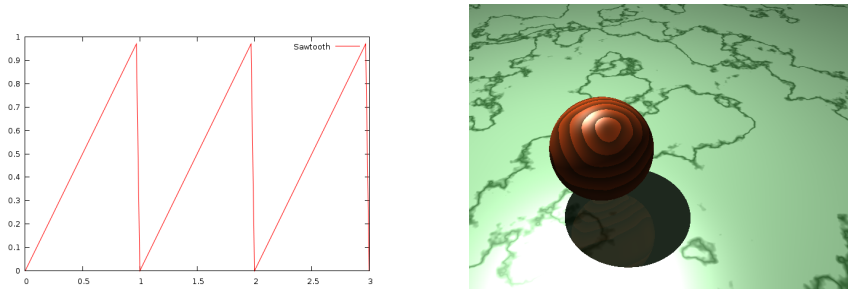

Figure 2: A sawtooth function (left) used to implement wood (right) by varying the blending degree between two materials depending on the radial distance.

Clicking the mouse enables a debugging output that is printed to the console. This will come in handy for you since you've just disabled all other debugging options. Check the source files (e.g., plane.cc) for examples of how to do these printouts.

Exercise 4.1 - create a new material

To determine the arguments used during the lighting computations for each pixel, the raytracer requests the lighting properties of the corresponding object at the 3D point corresponding to this pixel. This is done by a call to the `getLightingProperties` function which is then passed to the *material* that the object have been given (or inherited from).

Your task in this exercise is to create a new type of material. There already exists examples for uniformly coloured Phong-shaded materials (see class SimpleMaterial) and for materials derived from a Perlin noise function mapped by linear interpolation to corresponding Phong shading properties (see class MaterialMap).

The material that you should create in this exercise is a wood material. This can be accomplished by using two sets of parameters, light wood and dark wood, and by blending between the two of them so that the material will form concentric circles of light and dark wood when viewed down the z axis.

The wood should start out dark and grow smoothly lighter, drop sharply down to dark and begin going to light again (see Figure 2).

First compute the distance to the z axis:

$$r = \sqrt{p_x^2 + p_y^2}$$

Next, compute the blending factor to use by using a sawtooth function on $r \cdot c$ for some suitable constant c. The easiest way to implement a sawtooth function is to use the `fmod` operator from `math.h`, i.e.:

$$b = \mathrm{fmod}(r \cdot c, 1.0)$$

Finally, to compute the resulting colour, perform blending using b:

$$(1 - b) \cdot C_{\mathrm{dark}} + b \cdot C_{\mathrm{light}}$$

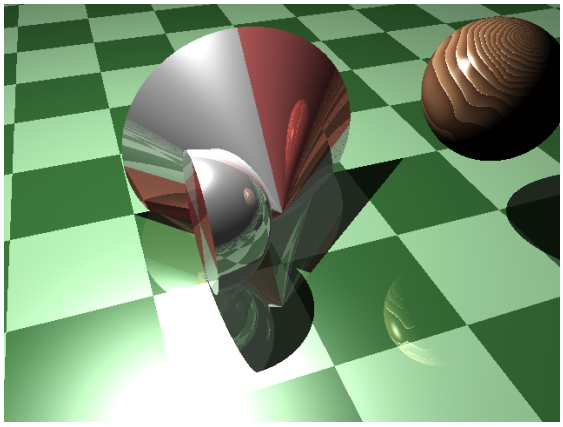

Figure 3: Adding a cone.

Apply this new material to one of the spheres and view it from different angles. Does the result look good? What happens when you change the constant c above? What are good lighting properties to use for the light and dark parts of the wood (e.g., amount of specular reflection, diffuse/ambient colours)?

As you may notice, the result is unrealistic since real wood does not have the same thickness for each year ring. Modify your expression so that the material has smaller rings further in, and so that there is a degree of random fluctuations from year to year. For the latter, you may want to use the one-dimensional noise functions given in `noise.h`.

Exercise 4.2 - create a new primitive object

The next exercise requires a bit more mathematics to solve. In this exercise you will add another type of primitive object, an infinitly large cone, to the raytracer. The basic definition of this cone is the following parametric function:

$$f(\vec{p}) = \vec{p}_x^2 + \vec{p}_y^2 - \vec{p}_z^2$$

The surface S of the cone is given by

$$S = \{\vec{p} : f(\vec{p}) = 0\}$$

This parametric definition will yield an inifitly large cone with its apex at the origin, growing in radius (x and y) as we step up the positive z axis. Due to the symmetry of the expression, it will also yield another cone meeting the first cone in the origin and growing in radius as we step down the negative z axis.

In order to implement this object in the raytracer you need to create a new object class and provide the three functions `lineTest`, `getNormal`, and `isInside` for it.

Exercise 4.2.1

The first of these functions, `lineTest`, requires a bit of math. To implement it you must analytically derive an expression for when a line starting at *origin* $\vec{o}$ with the given *direction* $\vec{d}$ intersects the object. The result should be the smallest α such that:

$$\vec{o} + \alpha\vec{d} \in S \qquad 0 < \alpha \leq m$$

where m is the *maximum distance* at which we are interested in intersections. If no such intersection exists then return m instead.

This becomes the same as solving the quadric equation:

$$f(\vec{o} + \alpha\vec{d}) = 0$$

Which can be simplified to:

$$(\vec{o}_x + \alpha\vec{d}_x)^2 + (\vec{o}_y + \alpha\vec{d}_y)^2 - (\vec{o}_z + \alpha\vec{d}_z)^2 = 0$$

Use this expression to derive an analytical expression for the *up to two* solutions that exists for α and return the first positive such solution, if any.

In order to elliminate one of the cones (e.g., the one extending down the negative z axis), you need to add a condition to the line test function so that it only returns solutions that correspond to a point on the positive z axis. *If a given solution α_1 or α_2 gives a z value on the negative z axis then discard that solution*.

For the second function, `getNormal`, you need to analytically derive an expression for the gradient of S for any given point $\vec{p}$. Ie, the vector consisting of the first-order derivatives with respect to x, y, z respectively.

$$\begin{pmatrix} \frac{df(\vec{p})}{dx} \\ \frac{df(\vec{p})}{dy} \\ \frac{df(\vec{p})}{dz} \end{pmatrix}$$

The third function, `isInside`, is perhaps the simplest function and is used for CSG (Constructive Solid Geometry) objects. A point $\vec{p}$ is inside if and only if $f(\vec{p}) \leq 0$. Remember to also check that a point is on the positive z axis as a requirement to be inside the cone.

Note that you can look inside the sphere.cc class for an examples on how these functions are done on spheres – which are, surprisingly enough, almost the same as a cones!

Exercise 4.2.2

Using the infinitly large cone in your previous exercise and some CSG (Constructive Solid Geometry) it is easy to create a cone of finite height. Do this by taking the intersection of your cone and a plane with a normal along the positive z axis and a suitable offset to create a cone of height 1. The result should look similar to Figure 3.

Another way of doing this is by adding an intersection with a plane already in the cone object – but that would be an entirely different exercise.

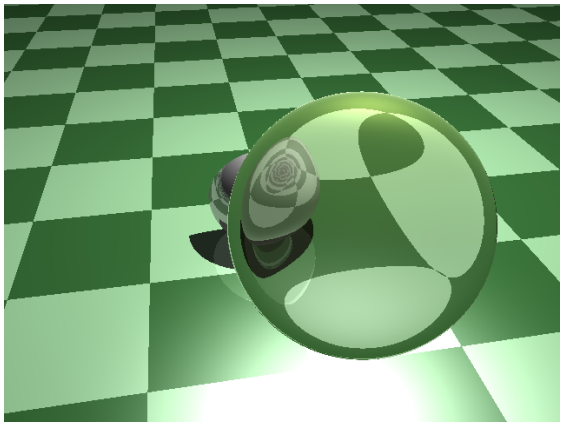

Figure 4: Refractive (transparent) objects.

Exercise 4.3 - implement transparency

This exercise is required for grade 5, but optional otherwise.

Add two new properties, *transparency* and *index_of_refraction*, to the lighting properties used by all materials and the basic raytracer. Let the raytracer use the returned transparency value to determine if it should cast another transparency ray through the object and add the resulting colour to the current pixel value. After adding proper transparency and refraction to one of the spheres, the result should look similar to Figure 4.

Do this with a similar check as to how it is determined if a reflection ray is cast (look at the accumulated value with which a ray contributes to the final pixel value) and cast a ray if the transparency is high enough.

The direction in which a transparency ray should be cast can be determined using the materials index of refraction as well as the *current* index of refraction (IOR). To do this properly, the index of refraction needs to be propagated through all calls to the raytracing function. However, to simplify things, you can also assume that you don't have nested refractive objects, so that the IOR is always 1.0 (vacuum) on one side of the surface.

To determine the direction in which a transparency ray is cast use Snell's law:

$$sin(\theta_1)/sin(\theta_2) = \eta_1/\eta_2$$

where θ_1 is the angle between the normal and the incoming vector and θ_2 is the angle between the normal and the outgoing vector.

Let $\vec{N}$ be the normal of the surface, $\vec{V_1}$ the incoming vector, η_1 incoming index of refraction,$\vec{V_2}$ the outgoing vector, η_2 the outgoing index of refraction. We then have:

$$\cos\theta_1 = \vec{N} \cdot (-V_1),$$

$$\cos\theta_2 = \sqrt{1 - (\frac{\eta_1}{\eta_2})^2(1 - (\cos\theta_1)^2)},$$

$$\vec{V_2} = (\frac{\eta_1}{\eta_2})\vec{V_1} + \left(\frac{\eta_1}{\eta_2}\cos\theta_1 - \cos\theta_2\right)\vec{N},$$

as long as $\vec{N} \cdot (-\vec{V_1})$ is positive – which means that the ray hits the object from the outside. If the ray hits the object surface from the *inside*, we need to invert the normal, because the normal that the raytracer gives us always points out from the object. In that case, we get

$$\vec{V_2} = (\frac{\eta_2}{\eta_1})\vec{V_1} - \left(\frac{\eta_2}{\eta_1}\cos\theta_1 + \cos\theta_2\right)\vec{N}.$$

You will need to check the direction $\vec{N} \cdot (-\vec{V_1})$ to know which expression to use for $\vec{V_2}$ and the two indexes of refraction.

If $\cos\theta_2$ lacks solutions or is outside the range $[-1, 1]$ then the transmission ray is not refracted but caught by the surface of the object.

If you want, you cad add handling of transparency also when computing shadows. In this case the effect of transparency only becomes a colouring of the light. Multiply the light colour with the transparent colour of all opaque object in a *straight line* from the point of interest to the light source. Note that it is impossible here to compute refracted shadows without resolving to photon maps or other advanced techniques.

Note that although the mathematics for this exercise may look intimidating, the amount of code needed to implement it is only about one page.

Examination

To pass the lab you must hand in a report showing the screen shots from your program and reporting on your implementation work and a discussion of the questions in the text. Make sure that in your reports, you include (copy/paste) important parts of your source code used to solve related problems. Code should be well commented and explained.

For grade 3 it is required that

- the report is well structured and written in clear English, and that the report is complete (given instructions in the text above) with all questions answered,
- all students in a group (pair) also orally can describe and argue for the method and the algorithms that have been used in each task.

For grade 4 it is also required that

- the report clearly describes the problem statement and what has been done in a way that can be understood by someone who knows the course contents but has not done the labs,
- the report critically reflects on what has been learned in the lab,
- the program code contains clear comments to explain it and that names of functions, variables and classes are understandable.

For grade 5 it is, in addition, required that

- the report demonstrates the ability to methodologically experiment with algorithm and parameter choices,
- and contains a solution of the additional assignment 4.3.

Deadline December 10