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)

30. October 2006

Computer Graphics I
Assignment 1

Submission deadline for the exercises: Thursday, 09. November 2006 before the lecture.

Hint: Send the source code to your "Bremser", in **one** zip-file with the following name: yourname_assignment01.zip. Written solutions have to be submitted in the lecture room before the lecture. Every assigment sheets counts 100 points plus some additional points from an $*$ exercise. You do not have to solve this one but you can get bonus points if you do it.

1.1 Primary Ray-Generation for a Perspective Camera Model (10 Points)

A *Perspective Camera Model* can be defined by the following parameters:

- Camera origin (center of projection) **pos**
- Viewing direction **dir**
- (Vertical) full opening angle *angle* of the viewing frustum (in degrees)
- Up-vector **up**
- Image resolution $resX \times resY$

Given the above camera description, derive the **ray**.**dir** for given pixel coordinates x, y (e.g. $128.5, 5.5$ through the center of the pixel). Pixels are squared and the projection plane is perpendicular to the **ray**.**dir**. Please incorporate the *acpect ratio* as well as the *focus* (distance from camera position to imageplane along **dir**).

1.2 Ray-Surface Intersection (10 + 20 Points)

Given a ray $\mathbf{r}(t) = \mathbf{o}+t\mathbf{d}$ with origin $\mathbf{o} = (o_x, o_y, o_z)$ and direction $\mathbf{d} = (d_x, d_y, d_z)$, derive the equations to compute the parameter t for the intersection point(s) of the ray and the following implicitly represented surfaces:

a) An infinite plane $(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$ through point $\mathbf{a} = (a_x, a_y, a_z)$ with surface normal $\mathbf{n} = (n_x, n_y, n_z)$, where any point $\mathbf{p} = (x, y, z)$ that satisfies the equation lies on the surface.

b) Given a sphere $(p_x - C_x)^2 + (p_y - C_y)^2 + (p_z - C_z)^2 = R^2 \Leftrightarrow (p - C)^2 = R^2$ with center $C = (C_x, C_y, C_z)$, radius $R \in \mathcal{R}$ and point $p = (p_x, p_y, p_z) \in \mathcal{R}^3$ on its surface. Compute the values of t for which the ray intersects the sphere.

You **have to** submit your solutions from this exercise in written form.

1.3 Implementation of a Minimal Ray Tracing System (10 + 40 + 10 Points)

Each basic ray tracing system in principle consists of only three simple parts:

- **Primary Ray Generation** for generating the rays to be cast from a virtual camera into the scene.
- **Ray Tracing** for finding the (closest) intersection of a ray with the scene to be rendered.
- **Shading** for calculating the 'color' of the ray.

In this exercise, you will build a **minimal** ray tracing system by implementing these three tasks. The CIP pools can be used to solve the exercises. To make this easier, you are provided with a basic ray tracing framework so that you just have to **fill in** the missing core parts.

- First, download the ray tracing framework from our course web site:

  `http://graphics.cs.uni-sb.de/Courses/ws0607/cg/microTrace01.zip`

- Uncompress this package and change to the `microTrace` directory:

  `unzip microTrace01.zip`

  `cd microTrace01`

- **After** you have filled in the missing parts, start the compilation by typing:

  `make`

- You can then start the ray tracer with:

  `./MicroTrace`

The provided ray tracing framework contains a number of useful C++ classes, which you will need for the practical exercises:

- A ready-to-use vector class `Vec3f`, which incorporates standard vector operations such as addition, subtraction, dot product, cross product, etc.
- In order to handle – and save – image data, a class `Image` is included that handles pixels of type `Vec3f`. Pixels are stored in RGB (Red, Green, Blue) format, where each color component ranges from 0.0 to 1.0. For example, black=$(0, 0, 0)$, white=$(1, 1, 1)$, red=$(1, 0, 0)$.

  With `Image::WritePPM(char *fileName)` image data is saved in PPM (Portable PixMap) format. See `MicroTrace.cxx` on how to use this class.

  The PPM file format to be used is a simple ASCII file format. Further information can for example be found at `http://www.dcs.ed.ac.uk/home/mxr/gfx/2d/PPM.txt` or type `man ppm` on UNIX systems. PPM files can best be viewed with the program `display`.

- A class `Ray`, where a ray is defined by its origin `Vec3f org`, its direction `Vec3f dir`, and its length `float t`.
- An **abstract** base class `Primitive` for handling scene primitives. For each primitive class derived from this base class (e.g. `Sphere or Triangle`), the pure virtual method `Primitive::Intersect(Ray &ray)` has to be implemented (see below).
- Furthermore an **abstract** base class `Camera` for handling the camera parameters. For each derived class (e.g. `PerspectiveCamera`), the pure virtual method `Camera::InitRay(float x, float y, Ray &ray)` has to be implemented. Whereby x and y specify the pixel coordinates which should be used to initialize the ray.

Before implementing anything read through the presented `classes` and `MicroTrace.cxx` and try to understand the object structure as well as internal dependencies. In this exercise your task now is to implement the missing parts in `MicroTrace.cxx`, `InfinitePlane.hxx`, `Sphere.hxx`, `Triangle.hxx`, and `PerspectiveCamera.hxx`:

a) Primary Ray Generation. Implement your camera model, you have derived in 1.1, into your micro ray tracer from the previous exercise (in `PerspectiveCamera.hxx`). Fill in the appropriate code into the `constructor` and `InitRay`-method. Test your camera implementation with the four corner rays of the image and camera model $c1$ (see MicroTrace.cxx). The *ray.dir* whould be: $(-0.5543, -0.415508, -0.721183)$, $(0.5543, -0.415508, -0.721183)$, $(-0.5543, 0.415508, -0.721183)$ and $(0.5543, 0.415508, -0.721183)$.

b) Ray Tracing. The heart of every ray tracer is to find the closest intersection of a ray with a scene consisting of a certain number of geometric primitives. This eventually requires to compute intersections between rays and those primitives.

For the three provided primitive classes `InfinitePlane` (in `InfinitePlane.hxx`), `Sphere` (in `Sphere.hxx`), and `Triangle` (in `Triangle.hxx`) implement the intersection methods `InfinitePlane::Intersect(Ray &ray)`, `Sphere::Intersect(Ray &ray)`, and `Triangle::Intersect(Ray &ray)` using the formulas you derived in exercise 1.2. For the ray-triangle intersection test you can use the method presented in the lecture or any other one.

The semantics of `Intersect` should be as follows:

- Return `true` if and only if a valid intersection has been found in the interval (`Epsilon`,`ray::t`). `Epsilon` is defined in `Vec3f.hxx`.
- If a valid intersection has been found with the primitive, set `ray::t` to the distance to this intersection point (if current $t < ray.t$).

Find the **closest** intersection of the ray with the scene by just testing all in `MicroTrace.cxx` defined primitives (`s1`, `s2`, `s3`, and `p1`) sequentially.

c) Shading. Just use constant colors for each primitive, and assign each pixel the color of the primitive the ray has hit. For this exercise, you can hard-code the color of the objects. To obtain the same image as below, use colors red (RGB=$(1,0,0)$) for `s1`, green (RGB=$(0,1,0)$) for `s2`, blue (RGB=$(0,0,1)$) for `s3`, yellow (RGB=$(1,1,0)$) for `p1`, cyan (RGB=$(0,1,1)$ for `t1`, and white (RGB=$(1,1,1)$) for `t2`.

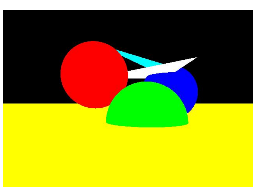
After finishing all implementation tasks in this assignment sheet render the example scene that is already hard-coded in `MicroTrace.cxx`. Render images (640×480 pixels) with the following camera definitions, and save them in “`perspective1.ppm`”, “`perspective2.ppm`” and ”`perspective3.ppm`”, respectively.

a) $\mathbf{pos} = (0, 0, 10)$, $\mathbf{dir} = (0, 0, -1)$, $\mathbf{up} = (0, 1, 0)$, $angle = 60°$

b) $\mathbf{pos} = (-8, 3, 8)$, $\mathbf{dir} = (1, -.1, -1)$, $\mathbf{up} = (0, 1, 0)$, $angle = 45°$

c) $\mathbf{pos} = (-8, 3, 8)$, $\mathbf{dir} = (1, -.1, -1)$, $\mathbf{up} = (1, 1, 0)$, $angle = 45°$

If your ray tracer works as expected, the first resulting image should look like this:



1.4 Ray Quadric Intersection (30 Points)*

Given a ray $R(t) = O + t \cdot D$ and quadric $ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j = 0$.

a) Compute the values t for which the ray intersects the quadric.

b) Derive the ray-sphere intersection formula from it, as a special case.

This is a pure theoretical exercise which has not to be implemented, but you can of course if you like.