UNIVERSITÄT DES SAARLANDES
Dr.-Ing. Hendrik P.A. Lensch
Lehrstuhl für ComputerGraphik
Art Tevs (tevs@mpi-inf.mpg.de)
Boris Ajdin (bajdin@mpi-inf.mpg.de)
Matthias Hullin (hullin@mpi-inf.mpg.de)

29. October 2007

Computer Graphics I
Assignment 1

Submission deadline for the exercises: Monday, 05. November 2007 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 + 15 + 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) An axis alignment bounding box represented by the vectors $\mathbf{min} = (x_{min}, y_{min}, z_{min})$ and $\mathbf{max} = (x_{max}, y_{max}, z_{max})$. Compute the values of t for which the ray intersects the bounding box and find the nearest one.

c) A triangle represented by three vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$. You have to find the value for t for which the ray intersects the triangle. Use either the method presented in the lecture or find another one which is suitable for a later implementation.

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

1.3 Implementation of a Minimal Ray Tracing System (10 + 30 + 5 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://www.mpi-inf.mpg.de/departments/d4/teaching/ws2007-2008/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`, `AABB.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 $ray.dir$ should 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`), `AxisAlignmentBoundingBox` (in `AABB.hxx`), and `Triangle` (in `Triangle.hxx`) implement the intersection methods `InfinitePlane::Intersect(Ray &ray)`, `AABB::Intersect(Ray &ray)`, and `Triangle::Intersect(Ray &ray)` using the formulas you derived in exercise 1.2.

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 $t < ray.t$).

Make sure your code is able to find the closest intersection of the ray with the scene. Do not assume that the order of primitives specified in `MicroTrace.cxx` provides a correct depth ordering.

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 `obj[0]`, yellow (RGB=$(1,1,0)$) for `obj[1]`, green (RGB=$(0,1,0)$) for `obj[2]`, cyan (RGB=$(0,1,1)$ for `obj[3]`, and white (RGB=$(1,1,1)$) for `obj[4]`.

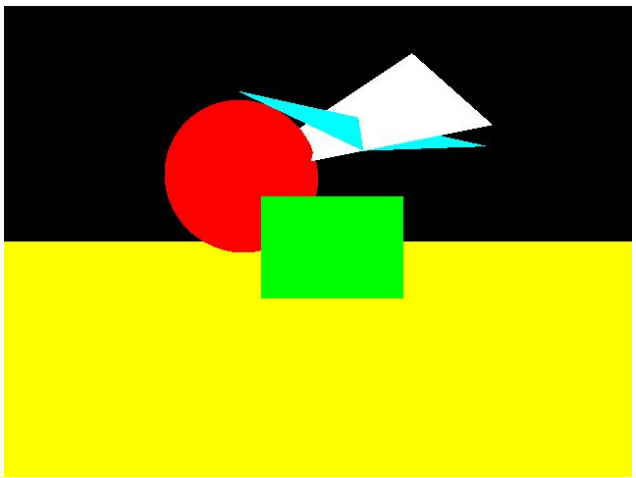
After finishing all implementation tasks in this assignment sheet render the example scene that is already hard-coded in `MicroTrace.cxx`. Render images (640×480 pixels) with the following camera definitions, and save them in "`perspective1.ppm`", "`perspective2.ppm`" and "`perspective3.ppm`", respectively.

a) **pos** $= (0, 0, 10)$, **dir** $= (0, 0, -1)$, **up** $= (0, 1, 0)$, $angle = 60°$

b) **pos** $= (-8, 3, 8)$, **dir** $= (1, -.1, -1)$, **up** $= (0, 1, 0)$, $angle = 45°$

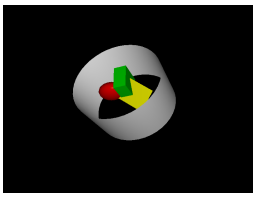
c) **pos** $= (-8, 3, 8)$, **dir** $= (1, -.1, -1)$, **up** $= (1, 1, 0)$, $angle = 45°$

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



1.4 Cylindric Camera Model (20 + 10 Points)*

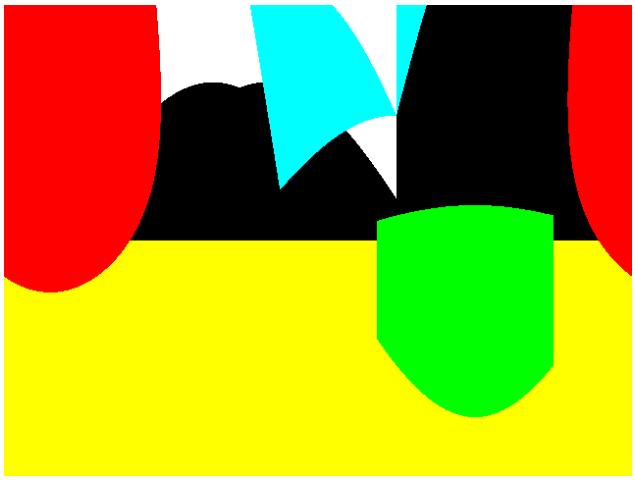
You have already learned a perspective camera model. In this model the image plane is assumed to be flat and each viewing ray has its starting position in the camera origin (center of projection).
In this exercise you will derive a cylindrical camera model. Imagine your image plane surrounds the scene and is represented by a cylinder.



We define the focal length **f** of the cylindrical camera model as the radius of the cylinder surrounding the scene geometry. The position of the camera **pos** represents the geometrical center of the image cylinder. The **up** vector represents the normal direction of the cylinder caps and can be assumed to be $\mathbf{up} = (0, 1, 0)$. The height of the cylinder is defined as **h**. The rays are traced from the image plane to the camera origin **pos**. The horizontal image axis (i.e. x-axis) is mapped around the cylinder.

a) Derive a primary ray-generation algorithm for the cylindrical camera model (similar to the Ex. 1.1).

b) Implement the derived algorithm and render the scene, defined in `MicroTrace.cxx` with the cylindric camera (image resolution is 640x480). The resulting image should look like this:



Parameters of the cylindric camera are: $\mathbf{pos} = (0, 0.5, 0)$, $\mathbf{up} = (0, 1, 0)$, $\mathbf{h} = 1.0$ and $\mathbf{f} = 0.25$.