UNIVERSITÄT DES SAARLANDES
Dr.-Ing. Hendrik P.A. Lensch
Max Planck Institut Informatik
Art Tevs (tevs@mpi-inf.mpg.de)
Boris Ajdin (bajdin@mpi-inf.mpg.de)
Matthias Hullin (hullin@mpi-inf.mpg.de)

5. November 2007

Computer Graphics I
Assignment 2

Submission deadline for the exercises: Monday, 12th November 2007

2.1 OBJ Scene loader (20 Points)

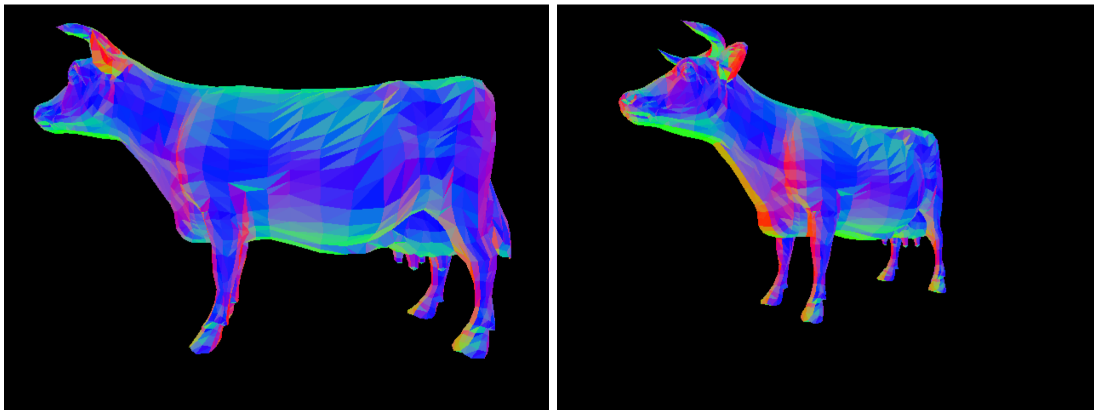
Until now we have only hard-coded our scene descriptions in `microTrace.cxx`. This is of course not practical. In the new framework, a method `ParseOBJ` is added to the class `Object`, in order to load a scene description from an `obj`-file. To make the method work proceed as follows:

a) Download the new microTrace framework from `http://www.mpi-inf.mpg.de/departments/d4/teaching/ws2007-2008/cg/microTrace02.zip`.

b) Have a look at the file `cow.obj`. Study how triangles are stored in the obj-format. The v's indicate a single 3D-vertex position, and the fs (faces) are indices to 3 vertex numbers a triangle consists of (please note that the face indices are starting with **1 and not 0**).

c) Implement the missing parts of the `ParseOBJ`-method.

Test your implementation with `cow.obj.` If your obj-importer works as expected you should see two resulting images of a cow like this:



To colorify the model we use the absolute values (x,y,z) of triangle's normal vector as a color value (r,g,b). **Hint:** The obj-file-format can be exported by various 3D-modelers. Nevertheless, the output might differ from modeler to modeler and there are also other tokens like *vn* for vertex normals or *vt* for texture coordinates. Check `http://www.royriggs.com/obj.html` for a full description of the format. For the rendering competition you may add some of these features.

2.2 Implementation of a kD-Tree (80 Points)

So far, your own ray tracer implementation has used no acceleration structure for reducing the number of ray/primitive intersections. This was simple to implement and worked relatively well. Unfortunately, this, of course, is not practical for larger scenes as you might have noticed in exercise 2.1 with the cow. As such, you need a data structure to speed up the process of finding the first hit of a ray with the primitives. Therefore, in this exercise you will implement a kD-Tree (a subclass of BSP-Tree). Proceed as follows:

a) A new class `Box` is now in the framework, which contains two `Vec3f`'s for the `min` and `max`-fields of the `Box`. Furthermore the class has a method `void Box::Extend(Vec3f)`. Implement the following functionality: If a is not inside a bounding box b, `b.Extend(a)` should extend the bounding box until it also includes a. **Tip:** Initialize your box with an 'empty box' ($min = +\infty$, $max = -\infty$). (5 Points)

b) The method `virtual Box Primitive::CalcBounds()` has to be implemented in every class derived from `Primitive`. (5 Points)

c) Implement the method `Object::CalcBounds()`, which should calculate the bounding box of the object's geometry. (5 Points)

d) A kD-Tree needs an initial test for ray/box intersection. This is required to check whenever a ray does hit the primitives at all. For this intersection test you can best use the *slabs* algorithm, and implement it in `Box::Intersect`. In case of a hit, the Intersect function should return a pair of entry and exit distance of the ray with the box. If there is no hit the value (FLT_MAX, -FLT_MAX) should be returned. (10 Points)

e) Implement the method `KDTree::FindSplitPlane(Node*)`, which should find a location of the splitting plane according to node's splitting axis. Use either median or average value of the primitive's position as spliting plane location. (10 Points)

f) Implement the method `KDTree::BuildTree(Node* node, Axis axis, int depth)`, which should set up the kd-tree by subdividing the nodes and putting the primitives into the corresponding child nodes. As soon as you have reached a maximum depth (e.g. 10), or you have less than a minimum number of primitives (e.g. 4), stop subdividing and generate a leaf node. Otherwise, split your current tree node into two child nodes and setup them recursively. The subdivision is started with a list of all primitives and an initial recursion depth of 0. Put the primitives based on the `FindSplitPlane()` method into the appropriate child node. If a splitting plane intersects the primitive, copy the primitive into both children. The tree should only contain the primitives in its leaf nodes. (20 Points)

g) For traversal (implement the `KDTree::Intersect()` and `KDTree::Node::LeafIntersect()`) use a recursive algorithm. Order the children of each node as near and far, based on ray's origin location in respect to the splitting plane of the current node. For this you have to implement the `KDTree::Node::GetNearFar()` method. Compute the signed distance of the ray and the splitting plane (`KDTree::Node::DistanceToSplitPlane()`) and use this value to decide which subtree to traverse next. For leaf nodes do intersect the ray with each primitive contained by the node to find the intersection point. (25 Points)

Use a simple, recursive algorithm, read Jim Arvo's *Linear-Time Voxel Walking for Octrees* to see how a traversal steps might be implemented or read the corresponding chapter in the thesis of Dr. Ingo Wald which can be found at `http://www.sci.utah.edu/~wald/PhD/index.html`. Search for more information on the net.

If your implementation was successfull you should encounter a big performance increase.
Instead of optimizing too much, rather concentrate on a stable, bug-free implementation.

2.3 Surface Area Heuristic in 2D* (20 Points)

The surface area heuristic is a local criteria to decide which splitting plane to use to split the current node during kd tree construction. In this exercise we consider only the two dimensional case.
Assume doing a ray triangle intersection has the cost C_I, and a traversal step the cost C_T. The currently processed kd tree node has the bounding box $B = ((B_{x0}, B_{y0}), (B_{x1}, B_{y1}))$ with the size $S_x = B_{x1} - B_{x0}$ and $S_y = B_{y1} - B_{y0}$ and contains N triangles T. For simplicity we describe the case of the splitting axis being the x-axis only, the second case is analogous and does not need to be considered for the exercise.
The cost of doing no split is $C_{nosplit} = N \cdot C_I$, as we have to intersect with each triangle during traversal.
We are interested in the best splitting position $d \in [B_{x0}, B_{x1}]$ which can be found by taking the d for that the following cost estimate is minimal:

$$C(d) = C_I \cdot (N_{left}(d) \cdot p_{left}(d) + N_{right}(d) \cdot p_{right}(d)) + C_T$$

The values N_{left} and N_{right} are the number of triangles overlapping with the left or right subbox and p_{left} and p_{right} are the probabilities that a uniformly distributed random ray hitting B hits also the left or right subbox. The probabilities can be computed by analyzing the surface area.

$$p_{left} = \frac{S_y \cdot (d - B_{x0})}{S_x \cdot S_y}$$

$$p_{right} = \frac{S_y \cdot (B_{x1} - d)}{S_x \cdot S_y}$$

Show that the cost function $C(d)$ can not be minimal at a position d where N_{left} and N_{right} are continous and $N_{left} \neq N_{right}$.