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)

November 13th, 2006

Computer Graphics I
Assignment 3

Submission deadline for the exercises: Thursday, 23th November 2006

3.1 OBJ Scene loader (20 Points)

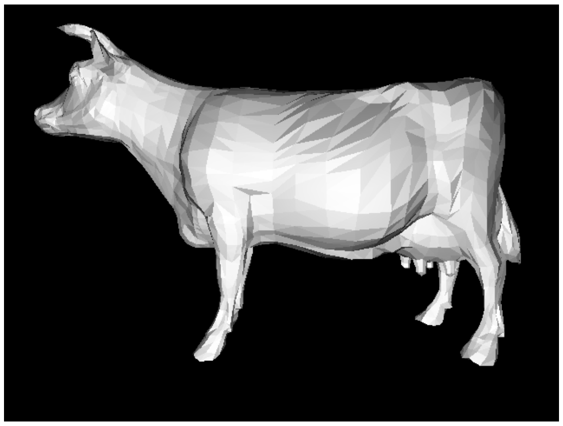
Until now we have only hard-coded our scene descriptions in `microTrace.cxx`. This is of course not practical. In the new framework, a method `ParseOBJ` is added to the class `Scene`, in order to load a scene description from an `obj`-file. To make the method work proceed as follows:

a) Download the new microTrace framework from `http://graphics.cs.uni-sb.de/Courses/ws0607/cg/microTrace03.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 *f*s (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 an image of a cow like this:



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.

3.2 Implementation of a Bounding Volume Hierarchy (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 3.1 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 Bounding Volume Hierarchy. 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' ($min = +\infty$, $max = -\infty$).

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

c) A Bounding Volume Hierarchy needs to test for ray/box intersections. 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.

d) Implement the method `Scene::CalcBounds()`, which should calculate the bounding box of the scene.

e) Implement the method `BuildTree(Box &bounds, vector <Primitive *> &prim, int depth)` of the class `BVH`. As soon as you have reached a maximum depth (e.g. 30), or you have less than a minimum number of primitives (e.g. 3 or 4), stop subdividing and generate a voxel. Otherwise, split your current box in the middle of the maximum dimension (this is a bad heuristic but simple to implement) and move the current primitives according to their center into the left or right list. Recursively call `BuildTree` with the respective box and vector for left and right children. In each node of the tree you have to store the box of the contained geometry for later traversal. Start subdivision with a list of all primitives, the total scene bounds, and an initial recursion depth of 0.

f) For traversal use a recursive algorithm. Order the children of each node as near and far, based on the signed distance of the entry point of the ray in the child's bounding box. Traverse the children recursively in that order, eventually skipping children that are not hit by the ray at all. If you find an intersection in the near child, which is closer to the origin of the ray than the entry point of the far child, skip the traversal of the far child.

3.3 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 ned 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 for 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 probability that a uniformly distributed random ray hitting B hits also the left or right subbox. The probabilities can be computed by an surface area analysis (or in 2D by a line length analysis):

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

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

Show that the cost function $C(d)$ has local minima only at positions where the number of triangles on the left (or right) change and are not equal. The functions N_{left} or N_{right} have discontinuities there.