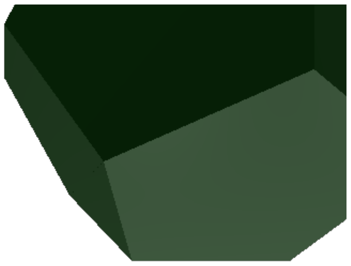
CSC305 A4
Chase Xu
V00924503

Q1. Ex.1: Triangle Mesh Ray Tracing [10pt]

Calculate the intersection of ray and triangle is like calculate the intersection of ray and parallelograms. The difference is use what condition to find whether ray is within the triangle. According to lecture slides, the constrain would be t > 0 and beta >= 0 and alpha >= 0 and g <= 1. I did not run for the dragon, because it will take too long without the BVH tree. The running result would be like following:



Q2. AABB Trees [20pt]

I have implemented a Tree in a recursive way. In theory the tree would keep dividing the nodes until it reaches the leaf. Then it will return the child nodes to upper level, and the upper level will extend the bbox and create a parent node as intended. The code is shown below:

```
Node build (const std::vector<Vector4d>& cents, const std::vector<Matrix3d>
&verts, const std::vector<AlignedBox3d>& boxes, std::vector<Node*> &nodes,
int parent_index=-1){

    if(cents.size() == 1){
        Node leaf;
        leaf.leaf = true;
        int index = cents[0](3);
        leaf.bbox = boxes[index];
        leaf.triangle = index;
        leaf.index = index;
        leaf.parent =  parent_index;
        Matrix3d v = verts[index];
        leaf.coordinates = v;
        nodes.push_back(&leaf);
        int inserted = nodes.size() - 1;
        return leaf;
    }
    int mid = ceil(cents.size()/2);
    std::vector<Vector4d> S1 = slice(cents, 0, mid-1);
    std::vector<Vector4d> S2 = slice(cents, mid, cents.size()-1);
    Node parent;
    AlignedBox3d box;
    parent.bbox = box;
    parent.triangle = -1;
    int size = nodes.size();
    parent.parent = size;
    nodes.push_back(&parent);
    parent_index = nodes.size()-1;
    //handle left nodes
    Node inserted = build(S1, verts, boxes, nodes, parent_index);
    parent.left =  inserted.index;
    Node left = *nodes[inserted.index];
    box.extend(left.bbox);
    //handle right nodes
    Node inserted2 = build(S2, verts, boxes, nodes, parent_index);
    parent.right = inserted2.index;
    Node right = *nodes[inserted2.index];
    box.extend(right.bbox);

    return parent;

}
```

Apparently, this function will return the root node, and this list of nodes will be save to AABBTree class; we could find each node according to its index.
According to the textbook, the code to detect ray and box intersection would be like below:

```
bool ray_box_intersection(const Vector3d &ray_origin, const Vector3d
&ray_direction, const AlignedBox3d &box)
{
    // TODO
    // Compute whether the ray intersects the given box.
    // we are not testing with the real surface here anyway.
    const Vector3d ray_direct =  ray_direction.normalized();
    // Vector3d t_xmin = ();
    Vector3d max = box.max();
    Vector3d min = box.min();
    const double width = max[0] - min[0];
    const double length = max[1] - min[1];
    const double height = max[2] - min[2];
    const double a = 1 / ray_direction.x();
    const double b = 1 / ray_direction.y();
    const double c = 1/ ray_direction.z();
    double t_xmin;
    double t_xmax;
    double t_ymin;
    double t_ymax;
    double t_zmin;
    double t_zmax;
    if(ray_direction.x() > 0){
        t_xmin =  a*(min.x() - ray_origin.x());
        t_xmax = a*(max.x() - ray_origin.x());
    }else{
        t_xmax =  a*(min.x() - ray_origin.x());
        t_xmin = a*(max.x() - ray_origin.x());
    }
    if(ray_direction.y() > 0){
        t_ymin = b*(min.y() - ray_origin.y()) ;
        t_ymax = b*(max.y() - ray_origin.y());
    }else{
        t_ymax = b*(min.y() - ray_origin.y());
        t_ymin = b*(max.y() - ray_origin.y());
    }
    if(ray_direction.z() > 0){
        t_zmin = c*(min.z() - ray_origin.z());
        t_zmax = c*(max.z() - ray_origin.z());
    }else{
        t_zmax= c*(min.z() - ray_origin.z());
        t_zmin= c*(max.z() - ray_origin.z());
    }

    if (t_xmin > t_xmax or t_ymin > t_ymax or t_zmin > t_zmax){
        return false;
    }else{
        return true;
    }
    // const Vector3d b = box.corner(BottomRightFloor<double,3>);
    return false;
}
```

Unfortunately, in find nearest object, I could not figure out the exact way to use bvh tree, so the exercise two is only half finished.

Q3. For all bounces

The bounces are easy, we could simply add all the parts we get from last assignment. And the results would be shown as below with maximum 3 bounces: