CS440 - Advanced Computer Graphics - Homework$_1$

Baris Sevilmis

March 14, 2022

1 Introduction

Purpose of this homework is to implement an efficient Ray Tracing algorithm using Nori framework in C++. In case of brute force search of each intersection of ray and triangles, rendering of the object requires unnecessarily long time and therefore would be inefficient to use in a realistic setting. Therefore, Octree data structure is used to distribute and store triangle indices into 3d bounding boxes. By simply using 3d bounding boxes in recursive order, we can trace the ray into particular positions, in which certain set of triangle are to be expected and could be checked for intersection instead of checking for all existing nodes.

In addition to the Octree data structure, a search algorithm is provided in which ray is traced to particular bounding boxes where intersection of ray and triangle could be found in recursive manner as well.
Octree has been dedicated its own class, where as build and search functions has been provided as a method to already existing Accel class.Following subsection will explain about the Octree class and search function.

2 Octree Data Structure and Traversing throught Octree

As mentioned in previous section, Octree requires a build and traverse functionality. OctreeNode class is provided to build and store the tree structure. Class contains 3 private variables. Index mask is used to store triangle indices from the mesh object of corresponding nodes, whereas curr_bbox stores the corresponding bounding box of triangles. Lastly, child_list pointer is utilized to store 8 children nodes. OctreeNode is a simple class with setters and getters to store and access bounding box and triangle indices.

buildRec() is one of our main objective functions, in which we create the tree structure. Base condition for leaf nodes is to have less than 10 triangle indices or stop after a certain depth which is 25 by default. Deeper the tree, faster is the search as we will have less triangles to search through. It has been decided through experimentation that 20 is a decent depth for our homework object. For the recursion part, we simply create new bounding boxes from current center point and corner points one by one, and check for intersections between triangle bound boxes and child bound boxes. Index masks are created for new bounding boxes and children node are set called recursively within the member function setChild() and connected to its parent. Index masks are kept just for the leaf nodes, otherwise memory requirement of the implementation would be unreasonable. Implementation of the function could be seen in following Code 1.
findIntersect() is our Octree traversal function, where we proceed by checking if current bounding box intersects the traced ray. We proceed by checking index masks and if node is a leaf node, check for intersection between triangles stored within the node. In case of bounding boxes not intersecting with the ray, we simply return false. If node is not leaf, then it means we have to recursively call findIntersect(). Instead of calling children nodes to find intersections does indeed work, but we can be faster if we sort children nodes by their distances to ray, we find intersections much faster. If an intersection is found, return value is true and propagated through boolean found value.
As you can see, we pass our objects by reference and clear index masks for all nodes except leaf nodes after we are done with them. This allows us to use low amounts of memory by simply referencing them. This allows us to modify intersection index, and other necessary variables within recursion easily as well, but maybe dangerous if not handled carefully.

```
OctreeNode* Accel::buildRec(const BoundingBox3f & bbox, std::vector<uint32_t> &indexMask, ↩
    uint32_t ctr){
    if(ctr >= DEPTH || index_mask.size() < 10){
      OctreeNode * newNode = new OctreeNode();
      newNode->setBoundingBox(bbox);
      newNode->setIndexMask(index_mask);
      return newNode;
    }
    else{
      OctreeNode * newNode = new OctreeNode();
      newNode->setBoundingBox(bbox);
      Point3f center = bbox.getCenter();
      BoundingBox3f child_bboxes[8] = {};
      for (int i = 0; i < 8; i++){
         child_bboxes[i] = BoundingBox3f(bbox.getCorner(i));
         child_bboxes[i].expandBy(center);
      }
      std::vector<std::vector<uint32_t>> child_triangle_indices(8);
      for(uint32_t i = 0; i < 8; i++) {
            for(uint32_t j = 0; j < indexMask.size(); j++)
                 if(child_bboxes[i].overlaps(m_mesh->getBoundingBox(indexMask[j])))
                    child_triangle_indices[i].emplace_back(indexMask[j]);}
      indexMask = std::vector<uint32_t>();
      for (uint32_t i = 0; i < 8; i++)
         if(child_bboxes[i].isValid())
            newNode->setChild(i,buildRec(child_bboxes[i],child_triangle_indices[i],++ctr));
      return newNode;}}
```

Code 1: OctreeNode* Accel::buildRec(BoundingBox3f, vector indexMask, ctr)

```
    bool Accel::findIntersect(const OctreeNode & node, Ray3f &ray, uint32_t &f, Intersection &↩
        its, bool shadowRay) const{
    bool found = false;
    BoundingBox3f bbox = node.getBoundingBox();
    if(!bbox.rayIntersect(ray))
        return false;
    std::vector<uint32_t> i_mask = node.getIndexMask();
    for(uint32_t i = 0; i < i_mask.size(); i++)
    {
        float u,v,t;
        if(m_mesh->rayIntersect(i_mask[i],ray,u,v,t) && t < ray.maxt)
        {
            if(shadowRay)
                return true;
            ray.maxt = its.t = t;
            its.uv = Point2f(u, v);
            its.mesh = m_mesh;
            f = i_mask[i];
            found = true;}}
    if(node.hasChild())
    {
        std::pair<float, OctreeNode*> sorted[8];
        for(uint32_t i = 0; i < 8 ; i++)
            sorted[i] = std::pair<float, OctreeNode*>(node.getChild(i)->getBoundingBox().↩
                distanceTo(ray.o), node.getChild(i));
        std::sort(sorted, sorted + 8, [ray](const std::pair<float, OctreeNode*> & left, const ↩
            std::pair<float, OctreeNode*> & right) {
            return left.first < right.first;});
        for(uint32_t i = 0; i < 8 ; i++)
        {
            found = found || findIntersect(*sorted[i].second, ray, f, its, shadowRay);
            if(shadowRay && found)
                return true;}}
    return found;
}
```

Code 2: bool Accel::findIntersect(const OctreeNode, ray, f, its, shadowray)

3 Results

This section provides experimentation details such as times required by build and traverse functions for different depth limitations and with sorting. Leaf node amount, internal node amount and average number of triangles per leaf node amount. For sake of simplicity, all experimentation are done with –threads 1 flag. Our rendering result is demonstrated within Figure 1.
Table 1 provides information for depth 20, 25 and 30. As it can be seen, deeper the tree, faster the traversal, and not much time is wasted checking each leaf node. Smaller bounding box means more less distance to ray, and therefore more accurate comparison for intersections. Since it was given that each leaf node should have equal or less than 10 triangles, we witness how much more efficient it is. Build time for different depths are not significantly different from each other and nothing much compared to the traversal. Main reason for traversal to be shorter for higher depth values is that we have to go through less triangles.

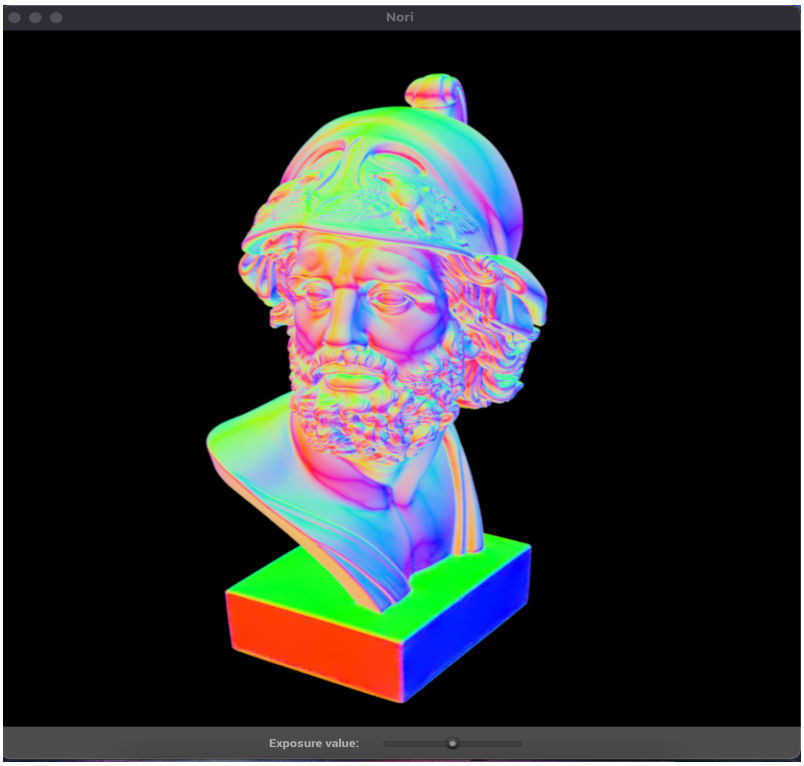

Figure 1: Rendered Ajax object

Table 1: buildRec() and findIntersect() statistics

–threads 1		*Depth = 20*	*Depth = 25*	*Depth = 30*
buildRec()	*build_time*	436 ms	563 ms	800 ms
	leaf_node	43499	160329	433728
	internal_node	12428	45808	123922
	avg_triangle	19.3006	8.16074	5.27198
findIntersect()	*std::sort*	3.2m	40.1s	22.8s