Acceleration Structures

KDTree

This functionality was implemented in src/scene/scene.cpp and src/scene/kdtree.h. There were three methods made: 1) Node::buildTree() and 2) Node::findBestSplitPlane() 3) Node::findIntersection(). In our implementation we had the following Node class.

```
//Parent class to SplitNodes and LeafNodes
class Node {
public:
   Node *left;
   Node *right;

   Node(){

   };

   //SplitPlane class
   class SplitPlane {
       public:
       int axis;
       double position;
       double left_count;
       double right_count;
       double left_area;
       double right_area;
       BoundingBox left_bb;
       BoundingBox right_bb;

       SplitPlane(int axis, double position) {
           this->axis = axis;
           this->position = position;
       };

   };
   //This node's boundary
   BoundingBox Boundary;
   bool findIntersection(ray& r, isect& i, double t_min, double t_max);
   Node buildTree(std::vector<Geometry*> objects, BoundingBox bb, int depth, int depthLimit, int
leafSize);
   SplitPlane findBestSplitPlane(std::vector<Geometry*> objList, BoundingBox bb);
   void printTree(Node *thisTree);

};
```

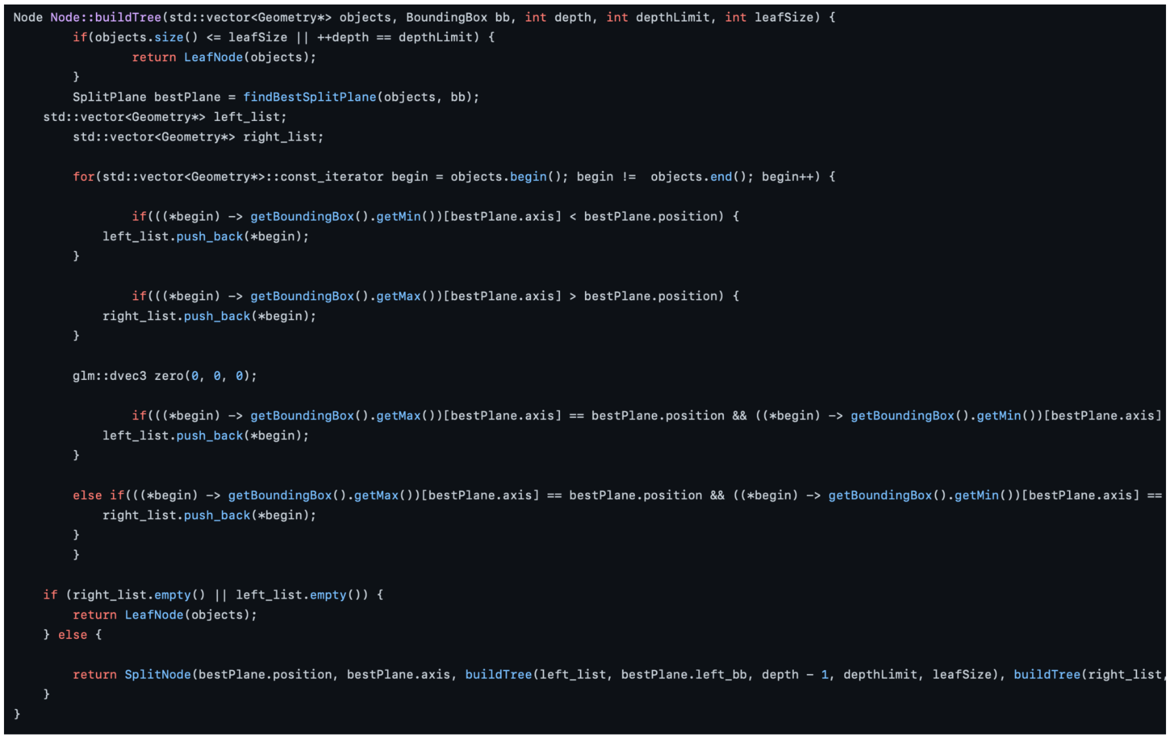
This Node class was parent to both a LeafNode and SplitNode class.

```
class SplitNode: public Node {
public:
   double position;
   double axis;


   SplitNode(double position, double axis, Node left, Node right) {
       this->position = position;
       this->axis = axis;
       this->left = &left;
       this->right = &right;
   };

   bool findIntersection(ray& r, isect& i, double t_min, double t_max);
};

class LeafNode: public Node {
public:
   std::vector<Geometry*> objList;
   LeafNode(std::vector<Geometry*> objList) {
       this->objList = objList;
       this->left = NULL;
       this->right = NULL;
   };
   bool findIntersection(ray& r, isect& i, double t_min, double t_max);
};
```

1) buildTree Implementation: recursive method that works as a space partitioning data structure for organizing nodes. A non-leaf node in KD tree divides the space into two parts (left subtree and right subtree). Using bounding box, depth, and leaf size, findBestSplitPlane to go through every object in the scene and adding it to the right subtree or right subtree
2) findBestSplitPlane implementation: this method used to extract the best plane to perform the next split. The way it works by going through all three dimensions for every object in the scene and store their positions in a candidate list and going through each candidate and extract leftObjectsCount, rightObjectsCount, leftBoundingBox, rightBoundingBox to get to the minSAM and therefore bestPlane
3) Intersect: in this method, we get the left and right bounding boxes using position, and just check which bounding box (left or right) the ray hits. We then recursively call this function until we reach a leafnode.

Code Snippets and Outputs:

Building the tree



```
Node Node::buildTree(std::vector<Geometry*> objects, BoundingBox bb, int depth, int depthLimit, int leafSize) {
        if(objects.size() <= leafSize || ++depth == depthLimit) {
                return LeafNode(objects);
        }
        SplitPlane bestPlane = findBestSplitPlane(objects, bb);
    std::vector<Geometry*> left_list;
        std::vector<Geometry*> right_list;

        for(std::vector<Geometry*>::const_iterator begin = objects.begin(); begin !=  objects.end(); begin++) {

                if(((*begin) -> getBoundingBox().getMin())[bestPlane.axis] < bestPlane.position) {
            left_list.push_back(*begin);
        }

                if(((*begin) -> getBoundingBox().getMax())[bestPlane.axis] > bestPlane.position) {
            right_list.push_back(*begin);
        }

        glm::dvec3 zero(0, 0, 0);

                if(((*begin) -> getBoundingBox().getMax())[bestPlane.axis] == bestPlane.position && ((*begin) -> getBoundingBox().getMin())[bestPlane.axis]
            left_list.push_back(*begin);
        }

        else if(((*begin) -> getBoundingBox().getMax())[bestPlane.axis] == bestPlane.position && ((*begin) -> getBoundingBox().getMin())[bestPlane.axis] ==
            right_list.push_back(*begin);
        }
        }

    if (right_list.empty() || left_list.empty()) {
        return LeafNode(objects);
    } else {

        return SplitNode(bestPlane.position, bestPlane.axis, buildTree(left_list, bestPlane.left_bb, depth - 1, depthLimit, leafSize), buildTree(right_list,
    }
}
```

Finding the best split plane

```
Node::SplitPlane Node::findBestSplitPlane(std::vector<Geometry *> objects, BoundingBox bb) {
  std::vector<SplitPlane> candidateList;
  for (int i = 0; i < 3; i++) {//iterate through x, y, z axis
      Node::SplitPlane p1(0, 0);
      Node::SplitPlane p2(0, 0);
      for ( std::vector<Geometry*>::const_iterator obj = objects.begin(); obj != objects.end(); obj++ )
{
          p1.axis = i;
          p1.position = ( (*obj)-> getBoundingBox().getMin())[i];
          p2.axis = i;
          p2.position = ( (*obj)-> getBoundingBox().getMax())[i];
          candidateList.push_back(p1);
          candidateList.push_back(p2);
      }
  }
  for (int i = 0; i < candidateList.size(); i++) {
      for ( std::vector<Geometry*>::const_iterator obj = objects.begin(); obj != objects.end(); obj++ )
{
          SplitPlane plane = candidateList[i];
          if(( (*obj)-> getBoundingBox().getMax())[plane.axis] <= plane.position) {
              plane.left_count++;
              BoundingBox temp = (*obj) -> getBoundingBox();
              plane.left_area += temp.area();
          }
          if(( (*obj)-> getBoundingBox().getMin())[plane.axis] > plane.position) {
              plane.right_count++;
              BoundingBox temp = (*obj)->getBoundingBox();
              plane.right_area += temp.area();
          }
          //Get left bounding box using our coordinate
          glm::dvec3 left_min = bb.getMin();
          glm::dvec3 left_max = bb.getMax();
          left_max[plane.axis] = plane.position;
          plane.left_bb = BoundingBox(left_min,left_max);
          //Get right bounding box using our coordinate
          glm::dvec3 right_min = bb.getMin();
          glm::dvec3 right_max = bb.getMax();
          right_min[plane.axis] = plane.position;
          plane.right_bb = BoundingBox(right_min, right_max);
      }
  }
  double minCost = INT_MAX;
  Node::SplitPlane bestPlane(0, 0);

  for (int i = 0; i < candidateList.size(); i++) {
      SplitPlane plane = candidateList[i];
       //SAM criteria: (chance of hitting left + channce of hitting right)/area
      double cost = (plane.left_count * plane.left_bb.area() + plane.right_count *
plane.right_bb.area()) / bb.area();
      if (cost < minCost) {
          minCost = cost;
          bestPlane = plane;
      }
  }
  return bestPlane;
}
```

Splitnode and leafnode's finding intersection

```
bool SplitNode::findIntersection(ray& r, isect& i, double t_min, double t_max) {
   //get left and right bounding boxes using position
   glm::dvec3 left_min = Boundary.getMin();
   glm::dvec3 left_max = Boundary.getMax();
   left_max[axis] = position;
   BoundingBox left_bb = BoundingBox(left_min, left_max);
   glm::dvec3 right_min = Boundary.getMin();
   glm::dvec3 right_max = Boundary.getMax();
   right_min[axis] = position;
   BoundingBox right_bb = BoundingBox(right_min, right_max);

   bool hits_left = left_bb.intersect(r, t_min, t_max);
   bool hits_right = right_bb.intersect(r, t_min, t_max);

      if (hits_left && !hits_right) {
          if (left->findIntersection(r, i, t_min, t_max)) { return true;}
      } else if (hits_right && !hits_left) {
          if (right->findIntersection(r, i, t_min, t_max)) {return true; }
      } else {
          if (left->findIntersection(r, i, t_min, t_max)) { return true; }
          if (right->findIntersection(r, i, t_min, t_max)) {return true; }
      }
  return false;
}
bool LeafNode::findIntersection(ray& r, isect& i, double t_min, double t_max) {
   for ( std::vector<Geometry*>::const_iterator obj = objList.begin(); obj != objList.end(); obj++ ) {
       isect c_i;
       if ((*obj)->intersect(r, c_i) && c_i.getT() >= t_min && c_i.getT() <= t_max) {
           i = c_i;
           return true;
       }
   }
  return false;
}
```

the debugging display:

Anti-Aliasing
This functionality was implemented in src/RayTracer.cpp. There was one modified method,RayTracer::tracePixel.
In this method we reset every pixel based on the given number of samples. We average out the colors for the surrounding pixels and return the new color.

Code Snippets and Outputs:

```
glm::dvec3 RayTracer::tracePixel(int i, int j)
{
       glm::dvec3 col(0,0,0);

       if( ! sceneLoaded() ) return col;
      double x = double(i)/double(buffer_width);
      double y = double(j)/double(buffer_height);
      unsigned char *pixel = buffer.data() + ( i + j * buffer_width ) * 3;
      double x_incr = (1.0/double(buffer_width))/samples;
      double y_incr = (1.0/double(buffer_height))/samples;
      for (double w = 0; w < samples; w++) {
              for (double h = 0; h < samples; h++) {
                      col += trace(x + w*x_incr, y + h*y_incr);
              }
      }
      col /= (double) (samples * samples);
      //cout << samples << endl;
      pixel[0] = (int)( 255.0 * col[0]);
      pixel[1] = (int)( 255.0 * col[1]);
      pixel[2] = (int)( 255.0 * col[2]);
      return col;
}
```

CubeMapping
-We successfully implemented this feature from Project 1

Issues during implementation

- **Our buildTree works, and we think our intersect functions works as well, but we weren’t sure where to place them as we traverse the tree**
- **Lots of C++ Struggles**

Known Bugs

- **N/A**

Future Work