https://gitlab.com/raymww/raytracing.git

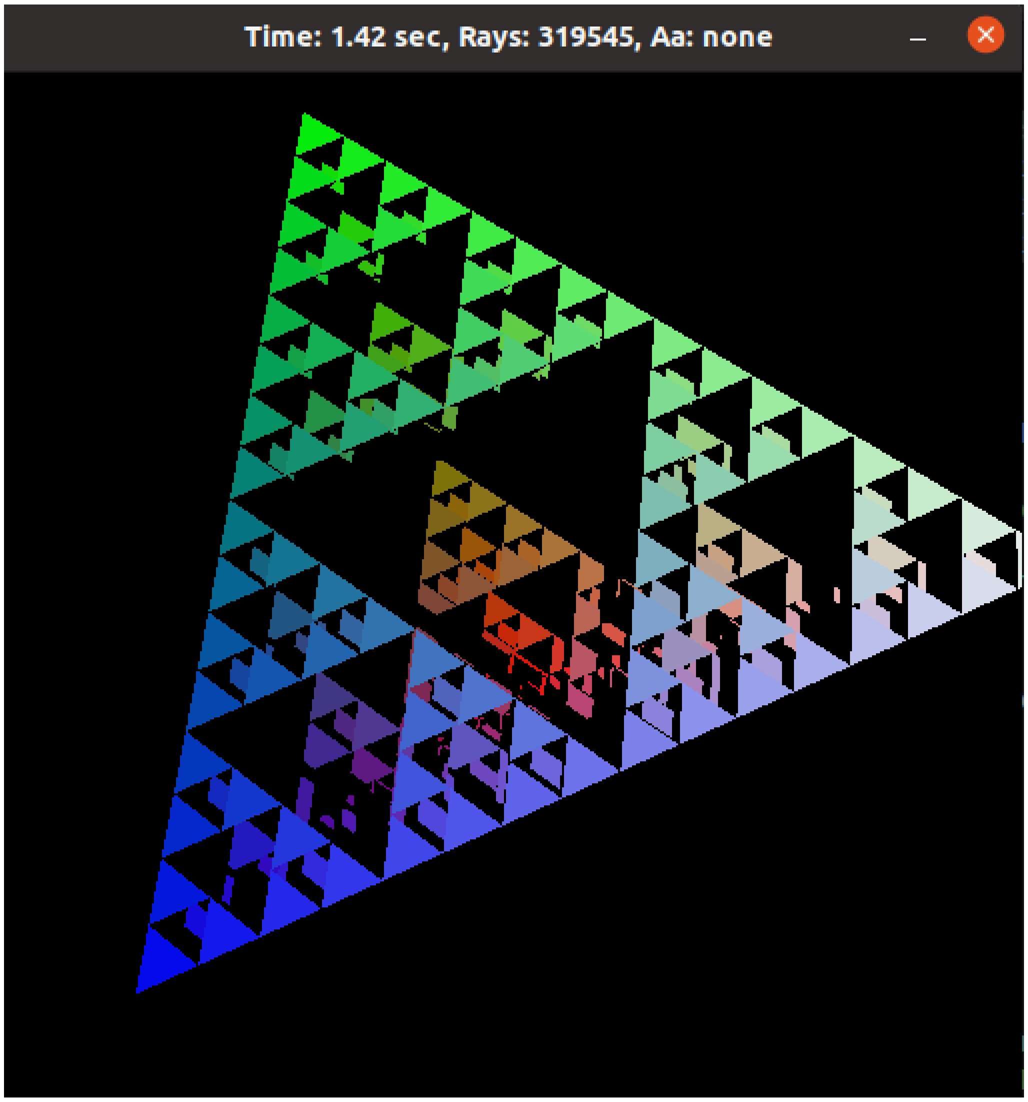
We (Max Hartfield and Raymond Wang) implemented all required functionality including Anti-aliasing and a KDTree with Spatial Data Structure (Geometry) and Spatial Data Structure (Trimesh)

Anti-aliasing:

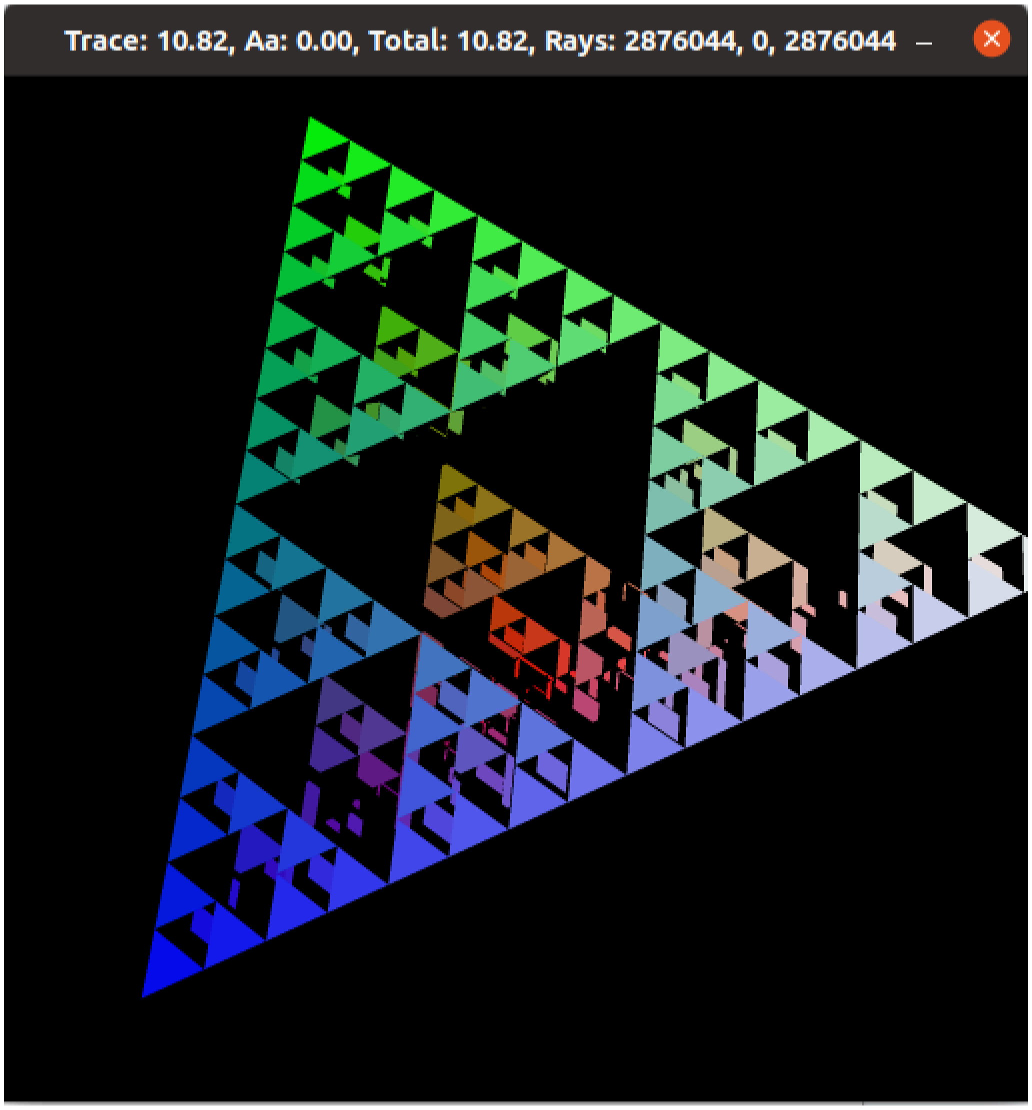
For Anti-aliasing, we first calculated the x offset and y offset based on the width/height of the pixel and the number of samples needed. Then, we moved x, y from the center to the top left ray that needed to be casted. Finally, we casted sample * sample rays in each position that is needed (sample rows, sample columns).

```
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;
  // decide to do AA or not
 if (traceUI->aaSwitch()) {
   // get the leftmost point to start casting aa rays
   double xOfs = 1.0 / ((double) buffer_width * (double) samples);
   double yOfs = 1.0 / ((double) buffer_height * (double) samples);
   double startX = x - (xOfs * samples / 2.0) + (xOfs / 2.0);
   double startY = y - (yOfs * samples / 2.0) + (yOfs / 2.0);

   // cast all AA rays needed through their respective points
   for(int k = 0; k < samples; k++) {
     for(int l = 0; l < samples; l++){
       col += trace(startX + (xOfs * k), startY + (yOfs * l));
     }
   }

   // average the color between all rays cast
   col = col * (1.0 / (double)(samples * samples));
 } else {
    // no aa, just cast one ray throgh center
    col = trace(x, y);
  }

  // set colors
  pixel[0] = (int)(255.0 * col[0]);
  pixel[1] = (int)(255.0 * col[1]);
  pixel[2] = (int)(255.0 * col[2]);
   return col;
}
```

Sier without Anti-aliasing


Time: 1.42 sec, Rays: 319545, Aa: none

Sier with Anti-aliasing



Trace: 10.82, Aa: 0.00, Total: 10.82, Rays: 2876044, 0, 2876044

Spatial Data Structure (Geometry):

For Spatial Data Structure (Geometry)'s KDtree, we first built the tree given a vector of objects, the entire scene's bounding box, and a depth. We followed the pseudo code for most of it excluding some optimizations and a return value. For the findBestPlane function we returned a vector of 2 doubles instead of a class, as all we need is the split axis (0-2) and the split value. When traversing the tree, instead of accessing a split plane class, we just look at the bbox in the children to determine the split plane. Then, we implemented the intersect method given a ray, a node, and a candidate list. The code for this section was very similar to the pseudocode for broad phase in the worksheet - we check if the ray intersects with the left or right, and recurse. Finally we initialized the kdTree and tweaked the scenes intersect method in scene.cpp.

Code:
.h file for the base kdTree class

```
class kdTree {
   public:
       kdTree *left;
       kdTree *right;
       BoundingBox bb;
       std::vector<Geometry*> objects;
       std::vector<TrimeshFace*> faces;

       // kd tree for geometry
       kdTree(std::vector<Geometry*> objects, BoundingBox bb, int depth);
       std::vector<double> findBestPlane(std::vector<Geometry*> objects,
BoundingBox bb);
       void intersect(ray &r, kdTree *node, std::vector<Geometry*>
&candList);

       // kd tree for trimeshes
       kdTree(std::vector<TrimeshFace*> faces, BoundingBox bb, int depth);
       std::vector<double> findBestPlane(std::vector<TrimeshFace*> faces,
BoundingBox bb);
       void intersect(ray &r, kdTree *node, std::vector<TrimeshFace*>
&candList);
};
```

constructor for KDtree with geometry

```
kdTree::kdTree(std::vector<Geometry*> objects, BoundingBox bb, int depth)
{
    this->bb = bb;
    this->left = NULL;
    this->right = NULL;
    this->objects = {};
    // base case, less than leaf size and depth
    if(objects.size() <= traceUI->getLeafSize() || depth + 1 ==
traceUI->getMaxDepth()) {
        this->objects = objects;
        return;
    }
    // find plane to split, best[0] = split value, best[1] = split axis
    std::vector<double> best = findBestPlane(objects, bb);
    // split the objects into the left and right based on the best plane
    std::vector<Geometry*> leftObjects;
    std::vector<Geometry*> rightObjects;
    for(Geometry *object : objects) {
        // min on axis is less than sv, so it is part of the left BB
        if(object->getBoundingBox().getMin()[best[1]] <= best[0]) {
            leftObjects.push_back(object);
        }
        // max on axis is greater than sv, so part of it must be in right
BB
        if(object->getBoundingBox().getMax()[best[1]] >= best[0]) {
            rightObjects.push_back(object);
        }
    }
    // early base case check
    if(leftObjects.empty() || rightObjects.empty()) {
        this->objects = objects;
        return;
    }
    //split into left and right bb based on axis and value from best
    glm::dvec3 leftMax = bb.getMax();
    leftMax[(int)best[1]] = best[0];
    BoundingBox leftbb(bb.getMin(), leftMax);
    glm::dvec3 rightMin = bb.getMin();
    rightMin[(int)best[1]] = best[0];
    BoundingBox rightbb(rightMin, bb.getMax());
    // recurse if case not reached
    this->left = new kdTree(leftObjects, leftbb, depth + 1);
    this->right = new kdTree(rightObjects, rightbb, depth + 1);
}
```

Findbestplane method:

```
std::vector<double> kdTree::findBestPlane(std::vector<Geometry*> objects,
BoundingBox bb) {
    std::vector<double> best(2);
    std::vector<std::vector<double>> planeList =
std::vector<std::vector<double>>();
    // add all possible split planes to planeList
    for(int i = 0; i < 3; i++) {
        for(Geometry *object : objects) {
            planeList.push_back({object->getBoundingBox().getMin()[i],
double(i)});
            planeList.push_back({object->getBoundingBox().getMax()[i],
double(i)});
        }
    }
    // sam should not be greater than this
    double minSAM = objects.size() * bb.area();
    // find optimal plane based on minimizing sam
    for(std::vector<double> plane : planeList) {
        //split into left and right BB based on the plane's axis and sv
        glm::dvec3 leftMax = bb.getMax();
        leftMax[(int)plane[1]] = plane[0];
        BoundingBox leftbb(bb.getMin(), leftMax);
        glm::dvec3 rightMin = bb.getMin();
        rightMin[(int)plane[1]] = plane[0];
        BoundingBox rightbb(rightMin, bb.getMax());
        int left = 0;
        int right = 0;
        // count num objects on either side of the plane
        for(Geometry *object : objects) {
            if(object->getBoundingBox().getMin()[(int)plane[1]] <=
plane[0]) {
                left++;
            }
```

```
           if(object->getBoundingBox().getMax()[(int)plane[1]] >=
plane[0]) {
               right++;
           }
       }
       // set best plane
       double SAM = left * leftbb.area() + right * rightbb.area();
       if(SAM < minSAM) {
           minSAM = SAM;
           best[0] = plane[0];
           best[1] = plane[1];
       }
   }
   return best;
}
```

Intersect function for broad phase testing:

```
// intersect a ray with the kdtree, do broad phase intersection test and
put all possible intersections in candList
void kdTree::intersect(ray &r, kdTree *node, std::vector<Geometry*>
&candList) {
   // base case: is leaf, and if it has objects, then add to candList
   if(!node->left && !node->right) {
     if(!node->objects.empty()){
       candList.insert(candList.end(), node->objects.begin(),
node->objects.end());
     }
     return;
   }
   double tMin;
   double tMax;
   node->bb.intersect(r, tMin, tMax); // sets tMin and tMax on BB
   // use the left and right child's bbox to calculate the split axis and
value.
   int axis;
   double sv;
   for(int i = 0; i < 3; i++){
        // if the left's bb max value on this axis is not the same as the
whole bbox, then
        // the box was split on this axis and at left's max bb val
        if(node->bb.getMax()[i] != node->left->bb.getMax()[i]) {
            axis = i;
            sv = node->left->bb.getMax()[i];
            break;
        }
    }
    // if the ray interescts with the left or right, keep recursing
    if(r.at(tMin)[axis] < sv || r.at(tMax)[axis] < sv) {
        intersect(r, node->left, candList);
    }
    if(r.at(tMin)[axis] > sv || r.at(tMax)[axis] > sv) {
        intersect(r, node->right, candList);
    }
}
```

Intersection testing using kdtree’s broad phase

```
bool Scene::intersect(ray &r, isect &i) const {
  std::vector<Geometry*> candList;

  // if the kd switch is on, then we will intersect using the kdtree's
  // candidates from broad phase testing, else, just do original brute
force
  if(traceUI->kdSwitch()){
    kdtree->intersect(r, kdtree, candList);
  } else {
    candList = objects;
  }

  double tmin = 0.0;
  double tmax = 0.0;
  bool have_one = false;

  // intersect with candList instead of objects (kd)
  for (const auto &obj : candList) {
    isect cur;
    if (obj->intersect(r, cur)) {
     if (!have_one || (cur.getT() < i.getT())) {
       i = cur;
       have_one = true;
     }
   }
 }
 if (!have_one)
   i.setT(1000.0);
 // if debugging,
 if (TraceUI::m_debug) {
   addToIntersectCache(std::make_pair(new ray(r), new isect(i)));
 }
 return have_one;
}
```

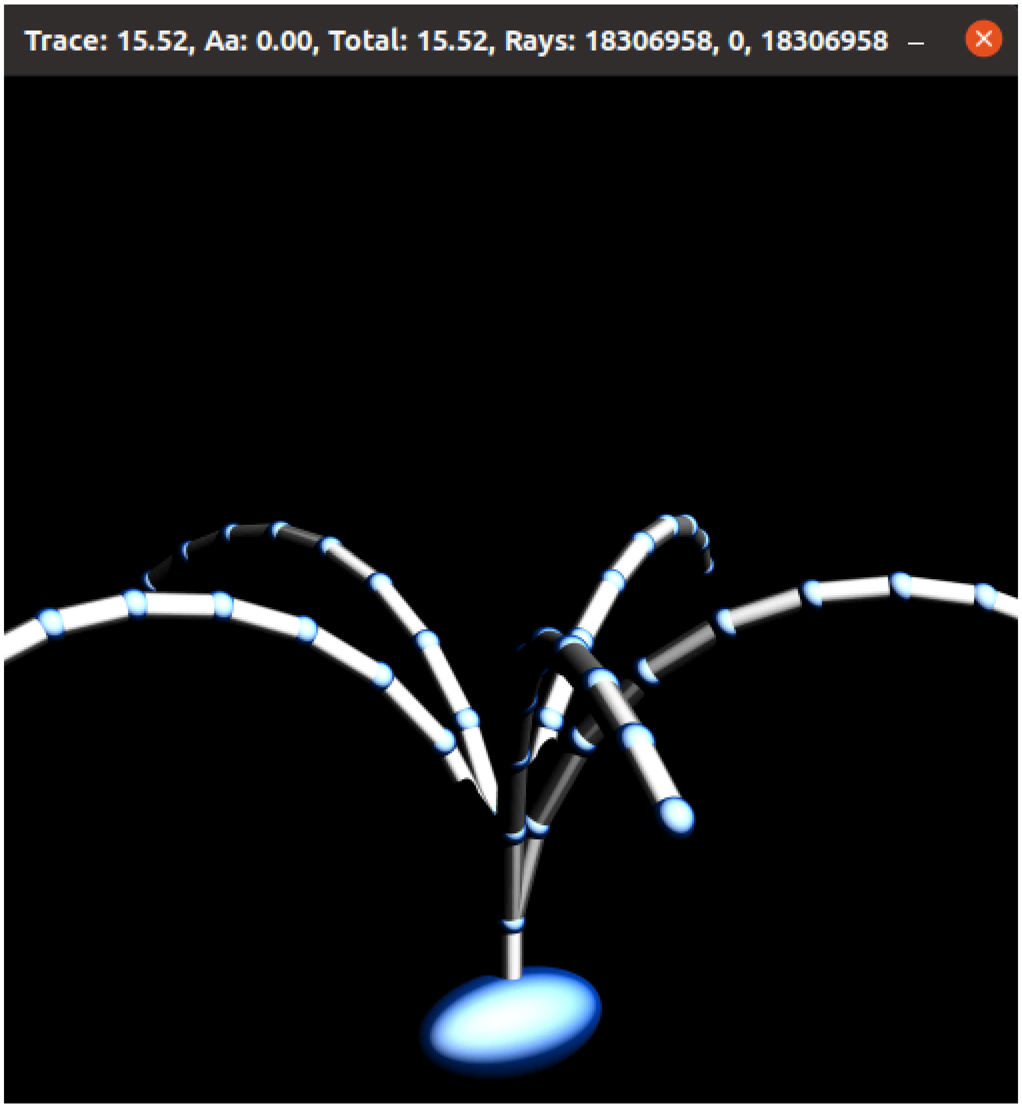
Initialization of kdtree in raytracer.cpp

```
 // YOUR CODE HERE
 // FIXME: Additional initializations
 // initialize kdTree after setup of scene
 if(traceUI->kdSwitch()){
   scene->initKdTree();
 }
```

initKdTree method

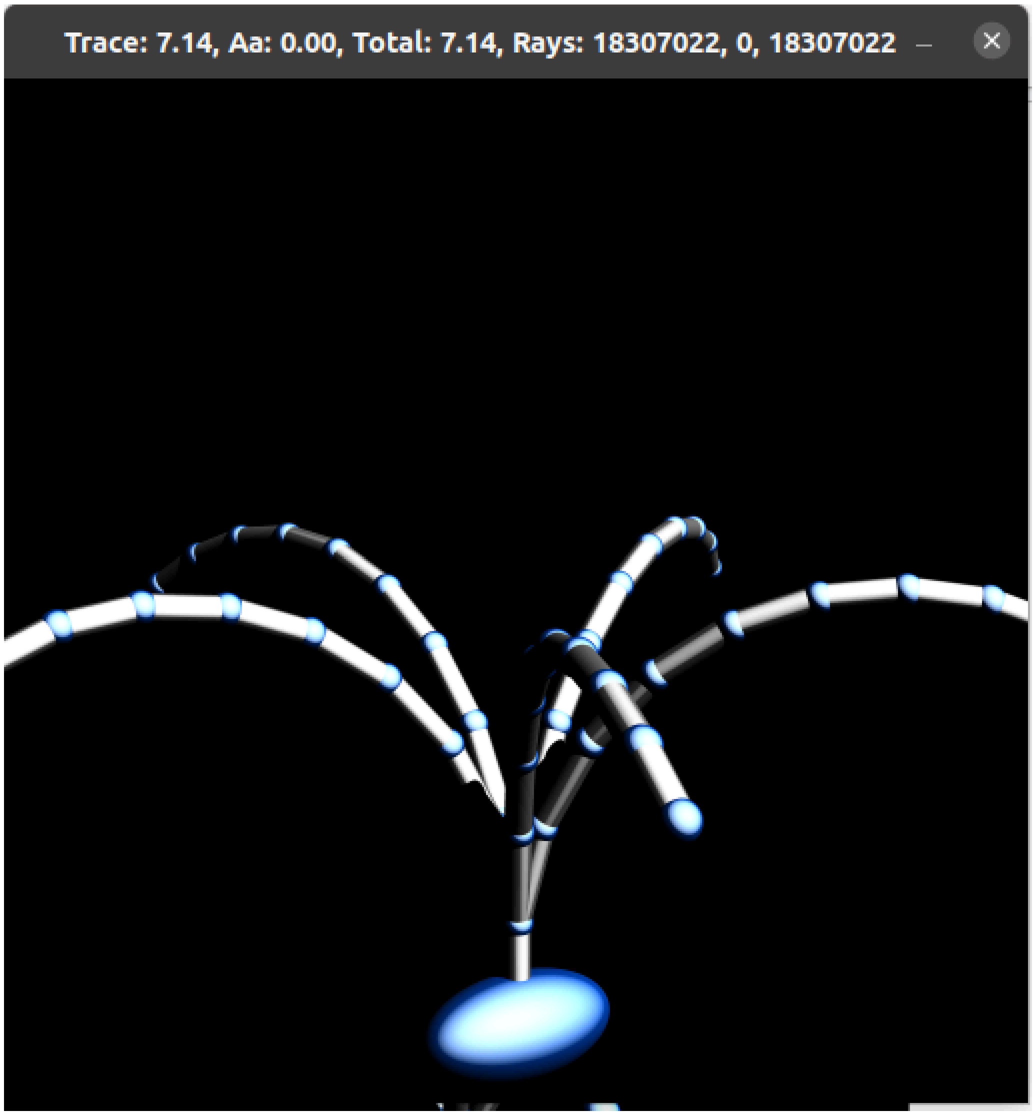
```
// set up KDTree for the scene
void::Scene::initKdTree() {
 std::vector<Geometry *> original = getAllObjects();
// if an object is a trimesh, then set up a kdtree for the trimesh's faces
 for(Geometry *object : original) {
   // check if is trimesh
   if(Trimesh* t = dynamic_cast<Trimesh*>(object)){
     t->tree = new kdTree(t->faces, bounds(), 0);
   }
 }
 // create kdTree
 kdtree = new kdTree(original, bounds(), 0);
}
```

Tentacles without kdTree and aa at 8



```
Trace: 15.52, Aa: 0.00, Total: 15.52, Rays: 18306958, 0, 18306958
```

Tentacles with kdTree and aa at 8


Trace: 7.14, Aa: 0.00, Total: 7.14, Rays: 18307022, 0, 18307022

Spatial Data Structure (Trimesh):

For Spatial Data Structure (Trimesh), we copy and pasted the same code for Spatial Data Structure (Geometry), but replaced all Geometry with TrimeshFace. We did this because we couldn't get templates to work, and still needed a kdTree for Trimeshes specifically

Code:

Constructor, Findbestplane, and Intersect are identical but with Geometry switched for TrimeshFace. Each trimesh has a kdtree, so when intersecting with the trimesh, it will then use the trimesh's kdtree to determine possible faces that intersect.

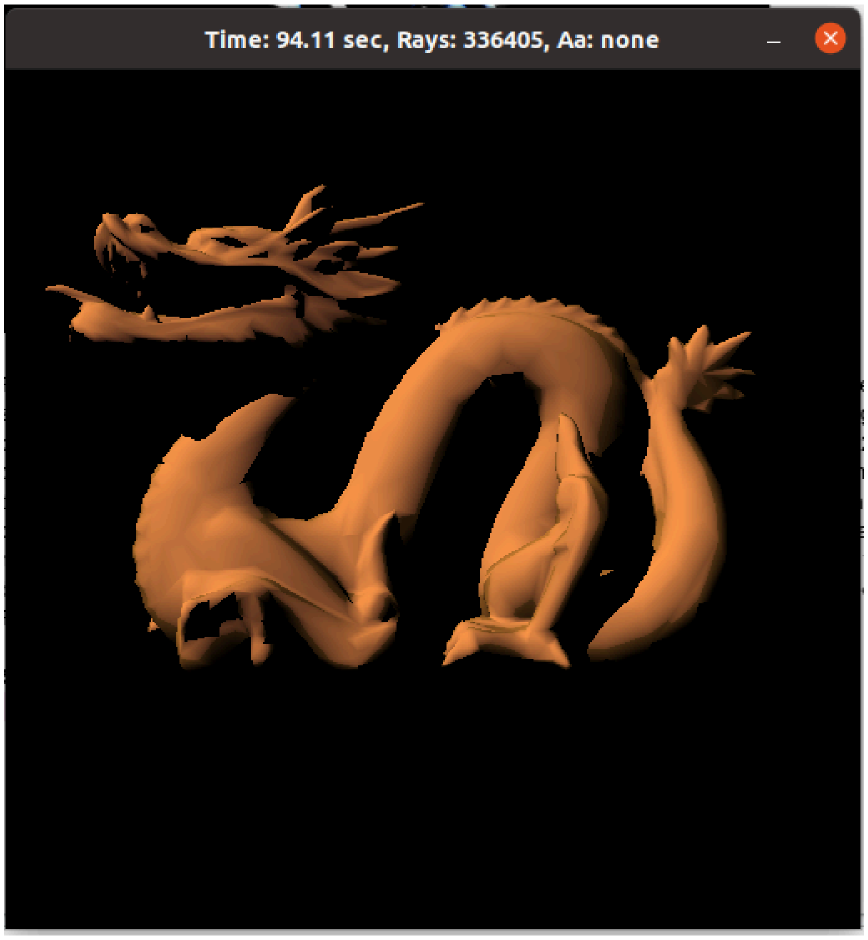
Kd-tree initialization for each trimesh:

```
// set up KDTree for the scene
void::Scene::initKdTree() {
 std::vector<Geometry *> original = getAllObjects();

 // if an object is a trimesh, then set up a kdtree for the trimesh's
faces
 for(Geometry *object : original) {
   // check if is trimesh
   if(Trimesh* t = dynamic_cast<Trimesh*>(object)){
     t->tree = new kdTree(t->faces, bounds(), 0);
   }
 }

 // create kdTree
 kdtree = new kdTree(original, bounds(), 0);
}
```

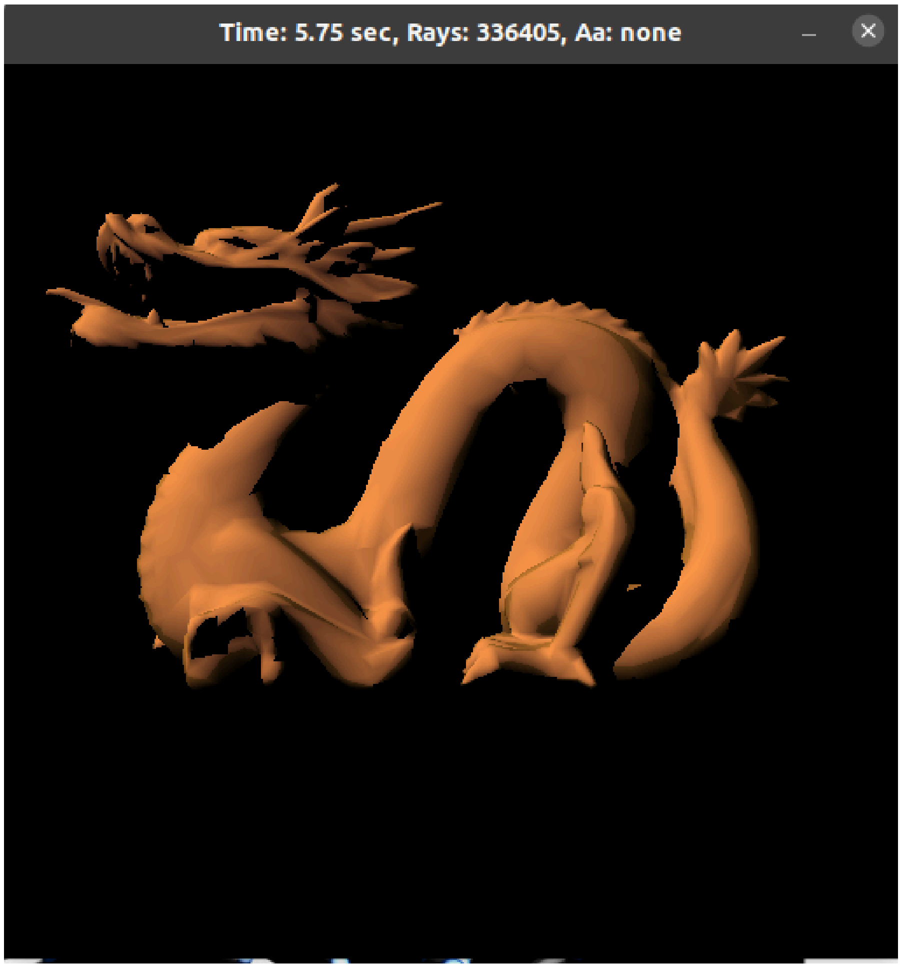
Trimesh intersection test using broad phase:

```
bool Trimesh::intersectLocal(ray &r, isect &i) const {
 bool have_one = false;
  // if kd switch is on, intersect will faces selected by kd broad phase
 // else just intersect with all faces. this is called from intersect in
 // the scene's kdtree and thus uses kdtrees for each trimesh obj
 std::vector<TrimeshFace *> candList;
 if(traceUI->kdSwitch()){
   tree->intersect(r, tree, candList);
 } else {
   candList = faces;
 }
  for (auto face : candList) {
   isect cur;
   if (face->intersectLocal(r, cur)) {
     if (!have_one || (cur.getT() < i.getT())) {
       i = cur;
       have_one = true;
     }
   }
 }
 if (!have_one)
   i.setT(1000.0);
 return have_one;
}
```

dragon without kdTree



dragon with kdTree


Time: 5.75 sec, Rays: 336405, Aa: none

Issues we encountered and fixed:

Anti-aliasing: We were trying to use aaImage since we saw it got called in CommandLineUI.cpp, but it wasn't actually getting called when we tested it. We fixed this issue by not using aaImage and fully implementing Anti-aliasing in tracePixel. It was also much cleaner this way since we didn't have to account for the two outer for loops

Spatial Data Structure (Geometry): We spent a lot of time debugging for one segfault. We fixed the error when we finally realized that one of the times when we were copying and pasting code for the left node we forgot to change the corresponding parts to the right node. It also took a long time to figure out which files needed kdTree and how to initialize them. At first we wanted to make a global kdTree and extern it in scene but we ended up with just giving scene its own instance of kdTree.

Spatial Data Structure (Trimesh): We were unable to get templates to work so we fixed this issue by copy and pasting the code for Spatial Data Structure (Geometry) and replacing all Geometry with TrimeshFace.

Known bugs: None.

Future work: None. We have fully implemented what was required for us in A1 and A2..

Optimizations Made: None.

Algorithmic Differences that result in differences from the solution: None.