karansher / **computer-graphics-bounding-volume-hierarchy** Public

forked from alecjacobson/computer-graphics-bounding-volume-hierarchy

Computer Graphics Assignment about Bounding Volume Hierarchies

0 stars 41 forks

Star Watch

Code Issues 28 Pull requests Actions Projects Security Insights

master

This branch is up to date with alecjacobson/computer-graphics-bounding-volume-hierarchy:master.

 **alecjacobson** update libigl ... on Feb 1 38

View code

Computer Graphics – Bounding Volume

README.md

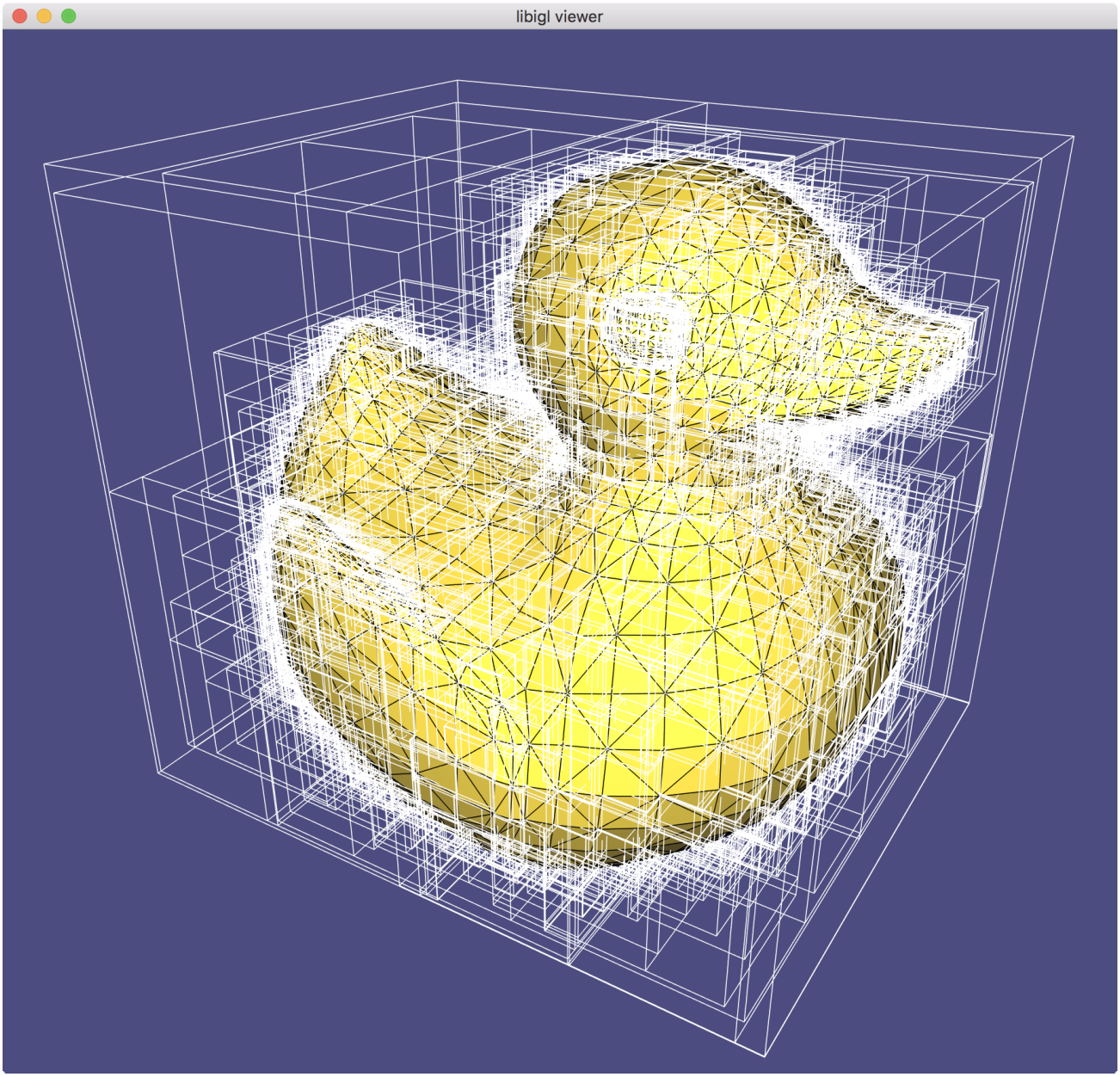
> **To get started:** Clone this repository and all its submodule dependencies using:
>
> ```
> git clone --recursive https://github.com/alecjacobson/computer-graphics-bounding-volume-hierarchy.git
> ```
>
> **Do not fork:** Clicking "Fork" will create a *public* repository. If you'd like to use GitHub while you work on your assignment, then mirror this repo as a new *private* repository: https://stackoverflow.com/questions/10065526/github-how-to-make-a-fork-of-public-repository-private

> **Note for Linux users:** if you're using Ubuntu, make sure you've installed the following packages if you haven't done so already:

```
sudo apt-get install git
sudo apt-get install build-essential
sudo apt-get install cmake
sudo apt-get install libx11-dev
sudo apt-get install mesa-common-dev libgl1-mesa-dev libglu1-mesa-dev
sudo apt-get install libxinerama1 libxinerama-dev
sudo apt-get install libxcursor-dev
sudo apt-get install libxrandr-dev
sudo apt-get install libxi-dev
sudo apt-get install libxmu-dev
sudo apt-get install libblas-dev
```

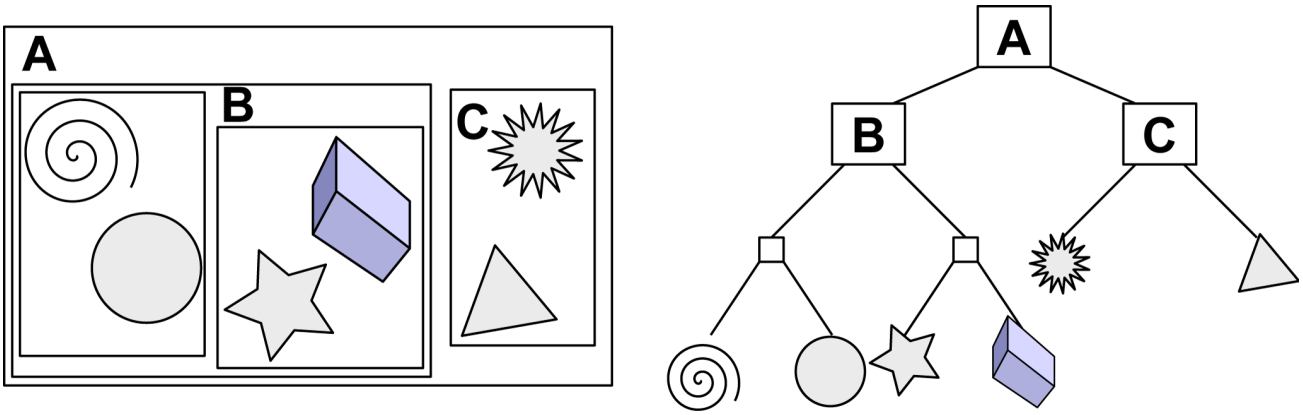
Background

Read Section 12.3 of *Fundamentals of Computer Graphics (4th Edition)*.

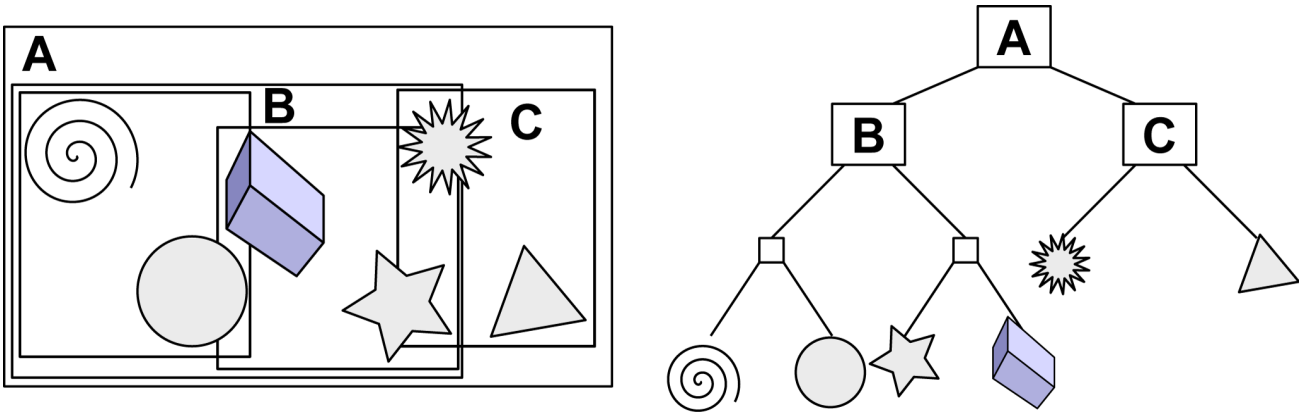


Object partitioning

In this assignment, you will build an Axis-Aligned Bounding-Box Tree (AABB Tree). This is one of the simplest instances of an *object partitioning* scheme, where a group of input objects are arranged into a bounding volume hierarchy.

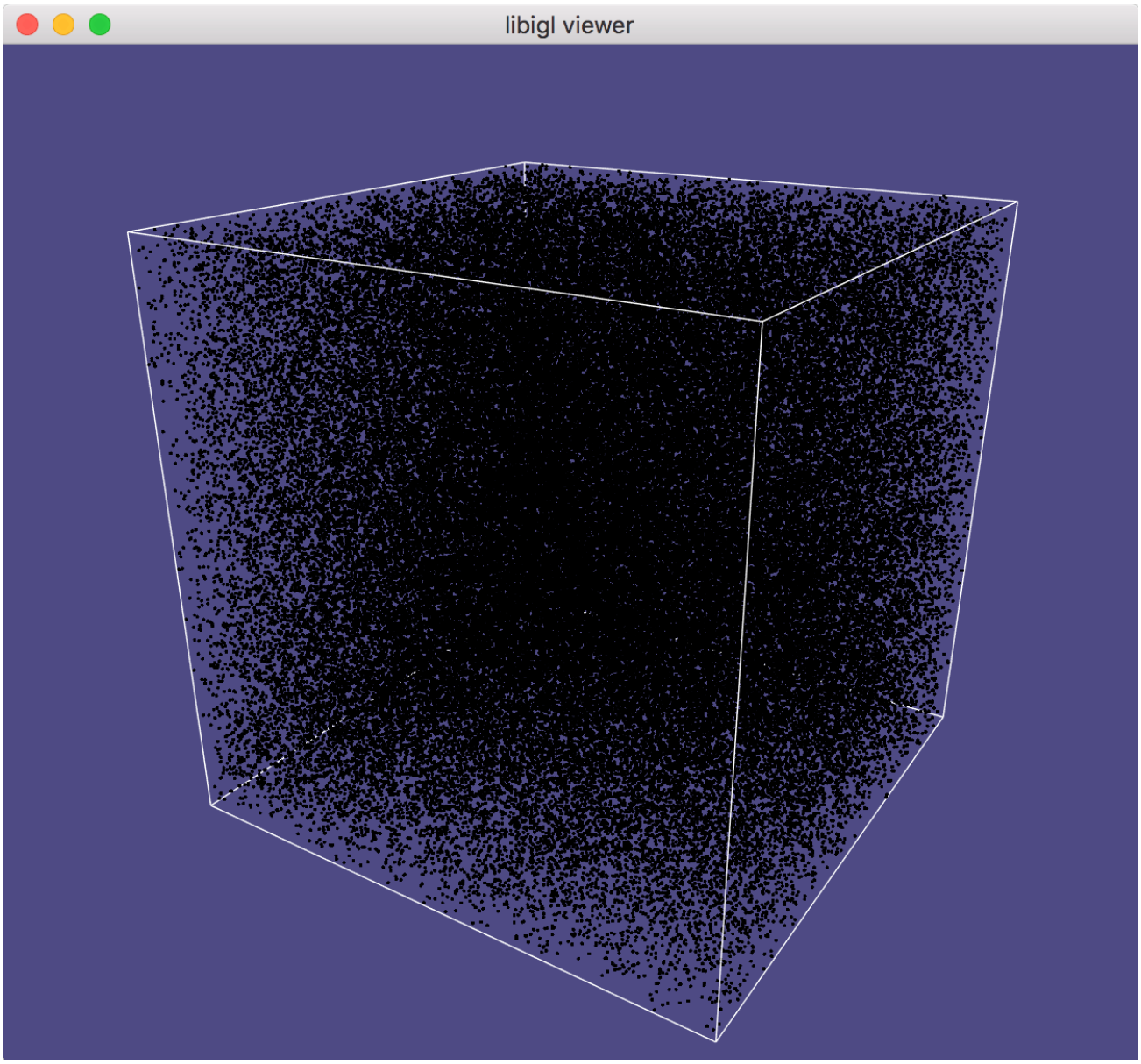


In our assignment, we will build a binary tree. Conducting queries on the tree will be reminiscent of searching for values in a binary search tree. However, objects in our tree will not be *perfectly* sorted. In general, the bounding boxes of "relatives" (even siblings) in our tree will overlap spatially.



By allowing bounding boxes to overlap we avoid the need to geometric split our objects.

> **Question:** If we use overlapping bounding boxes (i.e., no splitting) to build an AABB Tree , how many leaves will there be?

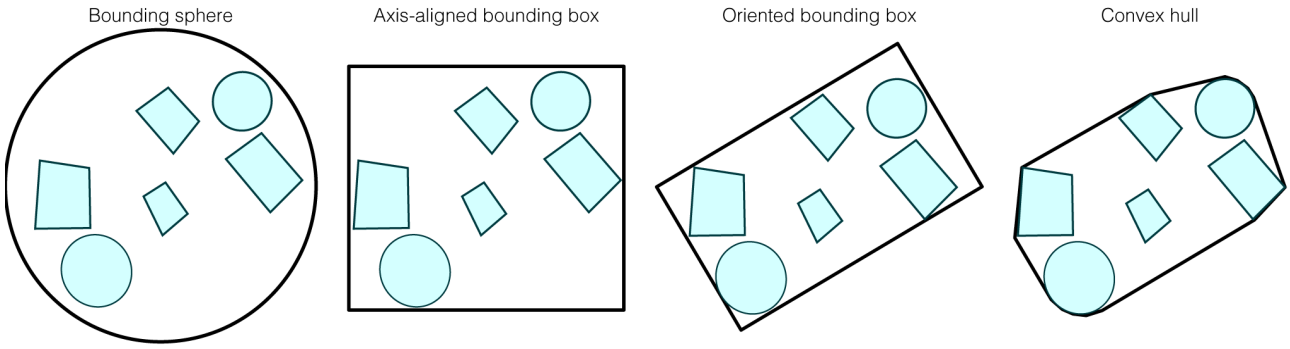


In contrast, space partitioning schemes (e.g., kd trees or octrees) divide space *perfectly* at each level of the tree, with no overlapping. This makes query code easy to write, but necessitates splitting of objects that inevitably straddle partition boundaries.

> **Question:** Which is better for an unstructured set of points, *space partitioning* or *object partitioning*?
>
> **Hint:** No perfect answer, but consider: do you ever need to split a point?

Bounding primitives

In this assignment, we will use axis-aligned bounding boxes (AABBs) to enclose groups of objects (e.g., points, triangles, other bounding boxes). In general, AABBs will *not* tightly enclose a set of objects. However, operations (e.g., growing the bounding box, testing ray-intersection or determining closest-point distances with an *axis-aligned* bounding box) usually reduce to trivial per-component arithmetic. This means the code is simple to write/debug and also inexpensive to evaluate.



Ray-intersection queries

See Section 12.3 of *Fundamentals of Computer Graphics (4th Edition)*.

Distance queries

The recursive algorithm in *Fundamentals of Computer Graphics (4th Edition)* for ray-AABBTree-intersection is essentially performing a depth first search. The search usually doesn't have to visit the entire tree because most boxes are not hit by the given ray. In this way, many search paths are quickly aborted.

On the other hand, using this style of depth-first search for closest point queries can be a disaster. Every box has *some* closest point to our query. A naive depth-first search could end up searching over every box before finding the one with the smallest query.

Are we just talking about worst-case complexity for pathological arrangements (e.g., a bunch of overlapping triangles piled at the origin)? No. Even on a well-balanced, minimally overlapping AABB tree we could end up exploring most of the leaves before finally finding the leaf containing the true closest point at the very end.

This implies that we can't just explore the left or right subtrees (or their progeny) in arbitrary order. A quick fix is to peek at the closet distance to the boxes containing the left and right trees respectively and prefer our depth first search in the closest direction. This helps, but we still end up *drilling* down to leaves when there are potentially entire large subtrees that are closer. The problem is that depth first search is inherently stack-based and we really want to use a priority queue to explore the current best looking path in our tree wherever it might be.

> **Question:** Hey! Where's the stack in depth first search? I implemented it using recursion, there's no `#include <stack>` in my code!
>
> **Hint:** Where are the instructions and data of your program stored?

Breadth-first search is a much better structure for distance queries on a spatial acceleration data-structure. Pseudo-code for a closest distance algorithm might look like:

```
// initialize a queue prioritized by minimum distance
d_r ← distance to root's box
Q.insert(d_r, root)
// initialize minimum distance seen so far
d ← ∞
while Q not empty
  // d_s: distance from query to subtree's bounding box
  (d_s, subtree) ← Q.pop
  if d_s < d
    if subtree is a leaf
      d ← min[ d , distance from query to leaf object ]
    else
      d_l ← distance to left's box
      Q.insert(d_l ,subtree.left)
      d_r ← distance to right's box
      Q.insert(d_r ,subtree.right)
```

> **Question:** If I have just a single query to conduct on a set of n objects, is it worth it to use a BVH?
>
> **Hint:** What is the complexity of *building* a BVH? What is the complexity of a single brute force query?

Intersection queries between two trees

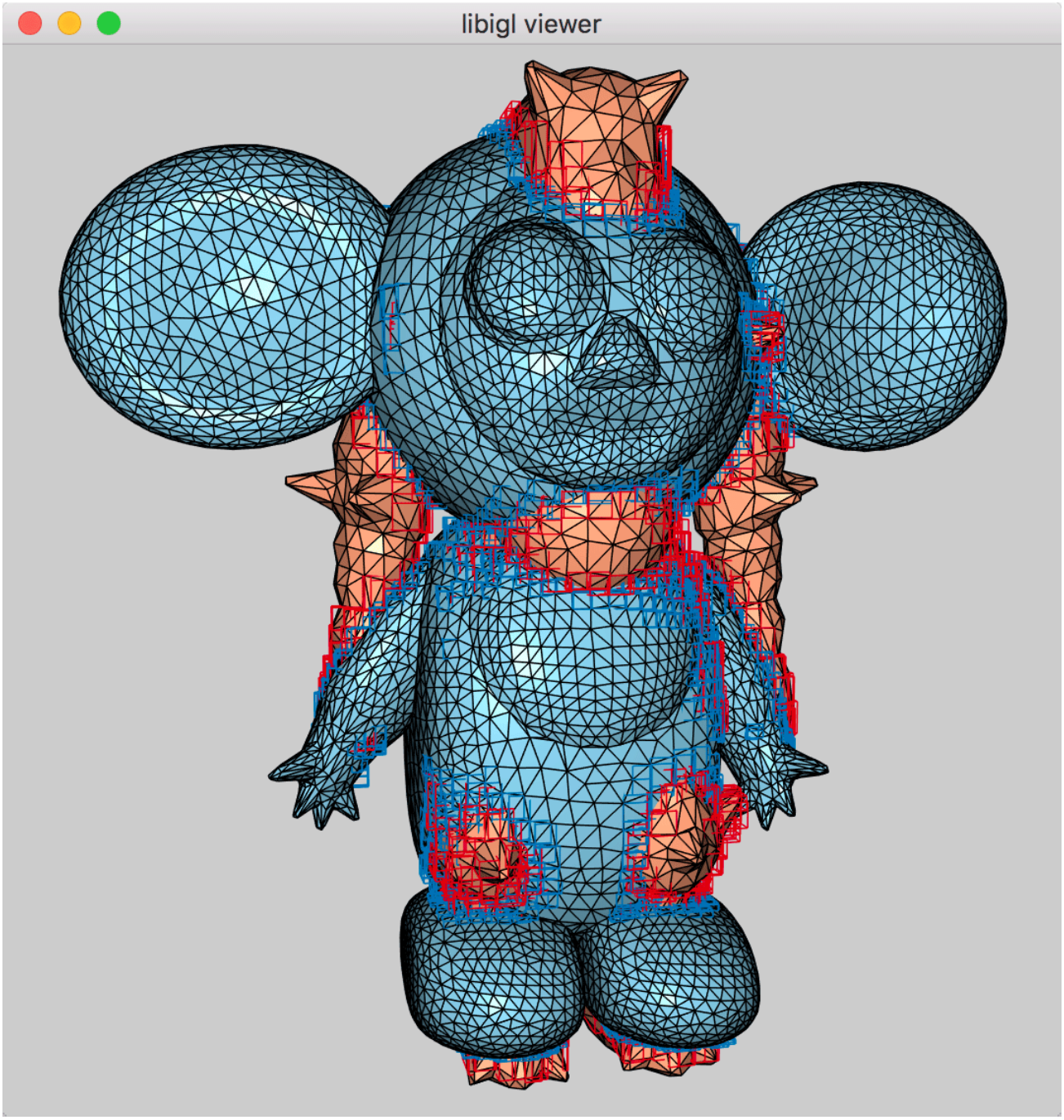
Suppose we want to find *all pairs* of intersecting triangles between two meshes. One approach would be to put one mesh's triangles in an AABB tree, then loop over the other mesh's triangles using the tree to accelerate intersection tests. This works well if the mesh in the tree has many more triangles than the other mesh, but can we do better if both mesh have many triangles? How about putting both meshes in a AABB trees. If the root bounding boxes don't overlap we find out *instantaneously* that there are no pairs of intersecting triangles. If they do overlap, we check their childrens' boxes against each other. Anytime two boxes don't overlap we save many expensive pairwise triangle checks. A rough sketch of this algorithm using a simple (i.e., non prioritized) queue is like this:

```
// initialize list of candidate leaf pairs
leaf_pairs ← {}
```

```
if root_A.box intersects root_B.box
  Q.insert( root_A, root_B )
while Q not empty
  {nodeA,nodeB} ← Q.pop
  if nodeA and nodeB are leaves
    leaf_pairs.insert( node_A, node_B )
  else if node_A is a leaf
    if node_A.box intersects node_B.left.box
      Q.insert( node_A, node_B.left )
    if node_A.box intersects node_B.right.box
      Q.insert( node_A, node_B.right )
  else if node_B is a leaf
    if node_A.left.box intersects node_B.box
      Q.insert( node_A.left, node_B)
    if node_A.right.box intersects node_B.box
      Q.insert( node_A.right, node_B)
  else
    if node_A.left.box intersects node_B.left.box
      Q.insert( node_A.left, node_B.left )
    if node_A.left.box intersects node_B.right.box
      Q.insert( node_A.left, node_B.right )
    if node_A.right.box intersects node_B.right.box
      Q.insert( node_A.right, node_B.right )
    if node_A.right.box intersects node_B.left.box
      Q.insert( node_A.right, node_B.left )
```

Careful, this sketch only considers a perfectly filled tree where nodes (and their left/right children) are never null pointers. Your trees may vary.

This ***broad phase*** identifies a set of overlapping bounding boxes containing one triangle each. The broad phase is quick because it uses the bounding volume hierarchy for acceleration and intersection between bounding boxes is a simple and fast. The list of candidate pairs scales with the number of *actual intersections* rather than the number of input triangles (as brute force double-for loops does). This list can then be processed using the (expensive) triangle-triangle intersection test in a ***narrow phase***.



> **Question:** Suppose we want to detect intersections for a simulation of two deforming meshes (e.g., elastic solids bumping into each other). Can we reuse our AABB Tree even if the meshes are deforming? What if they're just moving rigidly (rotations and translations)?
>
> **Hint:** Is an axis-aligned box still axis-aligned if it's rotated 45°?

Timing

Never conduct performance evaluations in debug mode. To set up a "release" mode version of your project use:

```
mkdir build_release
cd build_release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

In this assignment, we're aiming to improve the asymptotic complexity for the average case. We will not formalize the probability distribution of inputs, but instead consider uniformly random point clouds or real-world surface models. The AABB Tree algorithms should behave like $O(\log n)$ compared to brute force $O(n)$ algorithms. For large inputs the difference should be striking.

Tasks

Whitelist

You're encouraged to use the following

- `std::numeric_limits<double>::infinity()` and `-std::numeric_limits<double>::infinity()` in `#include <limits>` are often useful for initializing values before calculating a running minimum or maximum respectively.
- `std::priority_queue`
- `std::list` useful as a simple (non-priority) queue
- `std::pair` often useful to store key-value pairs (e.g., a priority and its corresponding object)

Shared Pointers

This assignment uses smart pointers. In particular, `std::shared_ptr`. For the most part you can use these like regular "raw" pointers. But for initialization use:

```
// Instead of:
// MyClass * A = new MyClass();
// Use
std::shared_ptr<MyClass> A = std::make_shared<MyClass>();
```

And omit deletion lines:

```
// No need for:
// delete A;
// Instead, it's destroyed when the last shared_ptr to A is destroyed
```

This assignment also uses inheritance. For example, `AABBTree` and `MeshTriangle` and `CloudPoint` are all derived from a common *base case* called `Object`.

Using `std::dynamic_pointer_cast<>`, it is possible to *attempt* to cast a `std::shared_ptr<>` to a base class instance into a `std::shared_ptr<>` of a subclass. This casting will only succeed if the underying instance actually is that subclass. Consider this self-contained example:

```
#include <memory>
#include <vector>
#include <iostream>

struct Object{/*Need a virtual function for polymorphism */virtual ~Object()
{}};
struct AABBTree     : public Object{};
struct CloudPoint   : public Object{};
struct MeshTriangle : public Object{};

int main(int argc, char * argv[])
{
  // Make a bunch of different subclasses of Object
  std::shared_ptr<AABBTree>     A = std::make_shared<AABBTree>();
  std::shared_ptr<CloudPoint>   B = std::make_shared<CloudPoint>();
  std::shared_ptr<MeshTriangle> C = std::make_shared<MeshTriangle>();
  // Put them in a list of Objects
  std::vector<std::shared_ptr<Object> > list_of_objects = {A,B,C};
  // Loop over each Object
  for(std::shared_ptr<Object> obj : list_of_objects)
  {
    // Attempt to cast to AABBTree
    std::shared_ptr<AABBTree> aabb = std::dynamic_pointer_cast<AABBTree>
(obj);
    // Test whether cast succeed
    if(aabb)
    {
      // Hooray. We can do AABBTree-specific operations on `aabb` now.
      std::cout<<"This object is an AABBTree."<<std::endl;
    }else
    {
      // Hooray. Now we know `obj` does _not_ point to an AABBTree. Hint,
hint.
      std::cout<<"This object is not an AABBTree."<<std::endl;
    }
  }
}
```

Compiling and executing this will print:

```
This object is an AABBTree.
This object is not an AABBTree.
This object is not an AABBTree.
```

Blacklist

Do not use or look at any of the following functions. Work out geometric derivations by hand rather than googling for a solution. Always cite online references as per academic honesty policies.

- `Eigen::AlignedBox`
- `igl::AABB`
- `igl::ray_box_intersect`
- `igl::ray_mesh_intersect`
- `igl::ray_mesh_intersect`

`src/ray_intersect_triangle.cpp`

Intersect a ray with a triangle (feel free to crib your solution from the ray casting).

`src/ray_intersect_triangle_mesh_brute_force.cpp`

Shoot a ray at a triangle mesh with n faces and record the closest hit. Use a brute force loop over all triangles, aim for $O(n)$ complexity but focus on correctness. This will be your reference solution.

`src/ray_intersect_box.cpp`

Intersect a ray with a ***solid*** box (careful: if the ray or `min_t` lands *inside* the box this could still hit something stored inside the box, so this counts as a hit).

`src/insert_box_into_box.cpp`

Grow a box `B` by inserting a box `A`.

`src/insert_triangle_into_box.cpp`

Grow a box `B` by inserting a triangle with corners `a`, `b`, and `c`.

`AABBTree::AABBTree` in `src/AABBTree.cpp`

Construct an axis-aligned bounding box tree given a list of objects. Use the midpoint along the longest axis of the box containing the given objects to determine the left-right split.

`AABBTree::ray_intersect` in `src/AABBTree_ray_intersect.cpp`

Determine whether and how a ray intersects the contents of an AABB tree. The method should perform in $O(\log n)$ time for a tree containing n (reasonably distributed) objects.

If you run `./rays ../data/rubber-ducky.obj` you should see something like:

```
# Ray Triangle Mesh Intersection
  |V| 334
  |F| 668

  Firing 100 rays...

  | Method      | Time in seconds |
  |:------------|----------------:|
  | brute force |   0.00158905983 |
  | build tree  |   0.00064301491 |
  | use tree    |   0.00004386902 |
```

If your method is incorrect, you will see some lines like this:

```
...
Error: #bf_hit(38) (1) != #tree_hit(38) (0)
...
```

This example line means that your brute force algorithm thinks ray 38 hits your object but your tree algorithm is not finding it.

`src/nearest_neighbor_brute_force.cpp`

Compute the nearest neighbor for a query in the set of n points (rows of `points`). This should be a ***slow reference implementation***. Aim for a computational complexity of $O(n)$ but focus on correctness.

`src/point_box_squared_distance.cpp`

Compute the squared distance between a query point and a box

`src/point_AABBTree_squared_distance.cpp`

Compute the distrance from a query point to the objects stored in a AABBTree using a priority queue. ***Note:*** *this function is* ***not*** *meant to be called recursively.*

Running `./distances 100000 10000` you should also see something like this:

```
# Point Cloud Distance Queries
    |points|: 100000
  |querires|: 10000

  | Method      | Time in seconds |
  |:------------|----------------:|
  | brute force |   1.50723695755 |
  | build tree  |   0.14633584023 |
  | use tree    |   0.05846095085 |
```

src/triangle_triangle_intersection.cpp

Determine whether two triangles intersect.

src/box_box_intersect.cpp

Determine if two bounding boxes intersect

src/find_all_intersecting_pairs_using_AABBTrees.cpp

Find all intersecting pairs of *leaf boxes* between one AABB tree and another

Running `./intersections ../data/knight.obj ../data/cheburashka.obj` will also produce something like this:

```
# Triangle Mesh Intersection Detection
  |VA| 2002
  |FA| 4000

  |VB| 6669
  |FB| 13334

  | Method      | Time in seconds |
  |:------------|----------------:|
  | brute force |   1.55577802658 |
  | build trees |   0.01804995537 |
  | use trees   |   0.00816702843 |
```

If your method is incorrect, you will see some lines like this:

```
...
Error: Intersecting pairs found using tree but not brute force:
  7,722
...
```

This indicates that your tree is finding *more* intersecting triangles than your brute force method. In particular, the tree thinks the 7-th triangle of mesh A is intersecting the 772-th triangle of mesh B.

Releases

No releases published

Packages

No packages published

Languages

C++ 60.8% **CSS** 23.5% **C** 12.0% **CMake** 3.7%