CS350
Assignment 3: Dynamic Aabb Trees and Frustum Culling

Submission:

Zip up and submit the entire project folder (the .sln and the CS350Framework folder). Make sure to not include any build artifacts! See the syllabus for submission specifications. Due March 11th (Sunday) by 11:55 pm.

Topics:

The purpose of this assignment is to construct and use a dynamic aabb tree as a general purpose high-performance spatial partition. This includes the usual optimizations of ray-casting, frustum-casting, and pair-queries. The aabb tree must also be used for frustum culling, that is not rendering objects outside the view frustum. In particular, temporal coherence should be used to efficiently reject an aabb based upon the last frame's results.

Testing:

You will be given a txt file containing sample results for the assignment. To test your project for this assignment run with the command line arguments of *"3 0*". The argument "0" runs all of the tests at once.
Do note that the graded tests will not be limited to the ones given to you!

Implementation Details:

Surface Area Heuristic: When choosing whether to traverse down the left or right side of the tree when inserting a node, choose the side that results in the smallest increase of surface area. If both nodes result in the same increase of surface area then choose the right side.

Debug Drawing: The height passed in corresponds to what level of the tree you should draw, where 0 is the root, 1 is the first set of children, and so on. If -1 is passed in draw the entire tree (including all internal nodes).

Insertion: When inserting a node into the tree use the surface area heuristic described in class. After inserting you must re-balance the tree using avl-rotations. When you split a leaf node put the old leaf node as the left child and the new node (the one you are inserting) as the right child. Also make sure to fatten the new aabb's half extent by the provided mFatteningFactor coefficient before inserting.

Update: If a node fully contains the new aabb then do not alter the tree structure. Otherwise remove then re-insert the updated node.

Removal: After removing the node from the tree you must apply a re-balance to the tree at the sibling of the node that was removed as well as all parents going up the tree. Don't forget that more than one balance can happen!

Balancing: When balancing the tree, you must use the avl rotation method described in class. There is one small ambiguity that can arise when a large and small child can't be distinguished (they are the same height). In this case my drivers choose the right node as the smaller (if left < right choose left). This case should be unlikely to happen. Note: the tree must be re-balanced after both insertion and removal. This is a recursive process where you walk up the tree testing each node from the insertion/removal point for an imbalance.

Ray-casting: You must recursively cast through the tree. If a node is not hit by the ray then no children should be tested. Return all client datas of leaf nodes that are hit in the tree.

Frustum-casting: If you did not already utilize "size_t& lastAxis" in the FrustumAabb test then you must do so now. This variable should be used to determine which plane axis to test first and should also be filled out with what axis causes the test to fail (if it does). You must use this when traversing your tree; for each node in the tree you should cache what the last axis was that caused a frustum cast to fail. In so doing, you can increase the chance to early out based upon last frame's results. This means that when performing a 2nd frustum cast against a scene where the root is off-screen the test should terminate with only 1 plane test to cull the entire tree. Similarly, if the root is fully contained (not intersecting) then all the leaf nodes should be included at a total of only 6 plane tests.

Pair Query: You must return all pairs in the tree without reporting any duplicates. You must do this utilizing the method describe in class (the 2 recursive functions). When you have two internal nodes and you must choose which one to split (which one's children to check) you should choose the one with the larger volume.

GetRoot: This function is used for me to investigate the structure of your tree. You should return the root node of your tree. Additionally you must implement the provided functions on the tree node for me to walk the structure of your tree.

Code Quality:

Code quality is a percentage modifier applied to your grade up to a -20%. Common code quality penalties are: redundant code, re-computed values, unnecessary allocations, and very hard to read code.

Grade Breakdown:

	Points	Percentage
DynamicAabbTree.cpp:		
Surface area heuristic	5	4%
Debug Drawing of tree levels	10	8%
Insertion	20	15%
Update	5	4%
Removal	20	15%
Rotation	20	15%
Ray casting	10	8%
Frustum casting (with temporal coherency)	20	15%
SelfQuery	20	15%
Total	130	100%

Extra Notes:

Avoid creating new files in the framework. I will be copying the relevant files for the assignment into a clean project for testing. For this assignment that is the files:

1. Geometry.hpp/cpp
2. Shapes.hpp/cpp
3. SimpleNSquared.hpp/cpp
4. DynamcAabbTree.hpp/cpp

The full framework should still be submitted though.

To make it easier to find missing functions they contain the comment "/******Student:Assignment3******/" that you can search.