Fun with BSP Trees (Huh?)

# Out: Monday, April 23 Due: Monday, April 30, 11:59PM

# Overview

In this assignment, you will be implementing a binary space partition tree. We’ve already implemented some functions that will help you with this, so you will be writing the following:

* BuildTree(List<GameObject> objects)
* DoAllIntersectingObjects(BoundingBox bbox, TreeFunction function)

As with the last assignment, we’ve provided test cases you can use to check your implementation using the automated testing tools. If you pass all the tests, the assignment is complete

# What is a BSP Tree and why should I care

Binary space partitioning is used in computer graphics and game development to implement efficient screen drawing and collision detection. For our purposes, we are writing a BSP tree to implement collision detection. A brute-force collision detection algorithm would look like this:

Foreach gameobject1 in gameobjects

Foreach gameobject2 in gameobjects

If gameobject1 is touching gameobject2

Do something

If you have n game objects, this algorithm takes time.

BSP trees operate in a similar fashion as regular binary search trees. However instead of splitting into subtrees based on whether values are greater or less than some number, we split based on whether objects are **in front of** or **in back of** a particular dividing plane:

|  |  |  |
| --- | --- | --- |
| **Component** | **Binary Search tree (integer key)** | **Binary Space Partitioning Tree** |
| Key | integer | Axis-aligned 3D plane |
| Greater than Comparison | > | Object in front of plane? |
| Less than Comparison | < | Object behind plane? |
| Equal to Comparison | == | Object intersects plane? |

Because of the structure of BSP trees, they have the same time complexities as binary search trees and are more efficient than the previously stated algorithm. To build a bsp tree, you sort the objects, find a plane that is in the middle of them (so that you have half your objects in front of the plane and half your objects behind the plane) and then call the build tree recursively algorithm on each half:

BuildTree(objects)  
 if objects.Count = =1  
 return new BSPTreeLeaf(objects[0])  
 Else  
 find the right splitting plane  
 return new BSPTreeInteriorNode(splittingPlane,  
 BuildTree(objects behind plane),  
 BuildTree(objects in front of plane))

The hard part is finding the plane that splits the objects in half. So we’ll have to try several different planes. Also, it may be that not plane will perfectly split the set of objects in half, so we’ll have to take the plane that gives us the best approximation to an even split, and keep track of which one is closest to splitting in half, then split on the best one we found.

Why do we want an even split? Because we want searching to be really fast. That means that, as with any binary tree, we want each step of the search to **eliminate half the objects**. For that to be true, the subtrees of a node need to be roughly the same size. If we can do that, then we have a balanced tree, and we get search times. If we don’t balance the tree, we could conceivably only eliminate one object with each plane, meaning we need planes rather than planes, and so our search takes time, which is much slower.

Finding a good splitting plane is hard to do. We’ll use a simplified case, where we only check for planes that are aligned with the X,Y, and Z axes. These axis-aligned BSP-trees are also called **kd-trees**.

# Getting started

This is a modified version of the last assignment. To get started:

* Unzip the assignment into a new directory
* Copy the files you modified for Assignment 2 (ListDictionary.cs and SyntaxTree.cs) into HappyFunBlob subdirectory (where the other HappyFunBlob files are).
* Open up the HappyFunBlob solution file
* In the HappyFunBlob project, open the BSPTree.cs file. This is where you’ll add your code for the first part of the assignment.

# New data types

This assignment uses some geometric data types from the XNA system that you haven’t seen before. For the most part, you don’t have to worry about them too much. We’ve done all the computational geometry for you so you don’t have to think about dot products or anything like that. Nevertheless, here are the new XNA data types you’ll encounter

* Vector3  
  Represents a position in space
* BoundingBox

Represents the approximate position and size of a GameObject. All you really need to know about these is that game objects have a property called BoundingBox and that Planes (see below) have a method called Intersect that tells you whether a bounding box is in front of, behind, or intersecting, the plane.

* Plane  
  Represents a plane in space. Once again, we’ve taken care of the geometric calculations for you. All you need to know is that planes have the following members:
  + Normal  
    The axis the plane points along
  + PlaneIntersectionType Intersects(BoundingBox b)  
    Determines whether a bounding box b is in front of, behind, or intersecting (straddling) the plane. Returns one of the three magic values:
    - PlaneIntersectionType.Front  
      The box is in front of the plane
    - PlaneIntersectionType.Back  
      The box is in back of the plane
    - PlaneIntersectionType.Intersecting  
      Some of the box is in front of the plane, and some is behind it

# The List<T> data type

You’ll also be using your first generic data type in this assignment, List<T>. List<T> is a resizable array[[1]](#footnote-1) of elements of type T. So you can think of List<T> as being just like an array of T (i.e. the type T[ ]), only it has some extra features. For the purposes of this assignment, you’ll only be using lists of GameObjects, so you’ll use the data type List<GameObject>. For this assignment, you’ll use the following operations on lists:

* *list*[*index*]  
  Returns the element at the specified position, just like an array.
* *list*.Count  
  Gives the number of elements currently in the list. Note that **this is different from arrays**, where you ask for the length using Length rather than Count.
* *list*.GetRange(int start, int length)  
  Copies a chunk of *list* and returns it. Returns a new List<T> consisting of elements *start* through *start*+*length*-1 of *list*.

# Overview of Starter Code

The first items to look at in the starter code are the bsp tree nodes which have been built from the base class BSPTreeNode:

* BSPTreeInteriorNode
  + Members:
    - BSPNode FrontSubTree
    - BSPNode BackSubTree
    - Plane SplitPlane
  + Functions:
    - DoAllIntersectingObjects(BoundingBox bBox, TreeFunction function)
* BSPTreeLeaf
  + Members:
    - GameObject GameObject
  + Functions:
    - DoAllIntersectingObjects(BoundingBox bBox, TreeFunction function)

In addition to the classes, we have provided several helper functions:

* void SortGameObjects(List<GameObject> gameObjects, Vector3 axis)
  + Sorts the list by the given axis aligned plane
    - For our purposes, there is the variable axes which is an array with the x, y and z unit planes respectively
* Plane SplittingPlane(GameObject o1, GameObject o2, Vector3 axis)
  + Returns a plane pointing along axis, and halfway between the two GameObjects
* bool SplitsAt(List<GameObject> objects, int splitIndex, Plane candidate)
  + Tests to see if the candidate plane works as a splitting plane (no intersections)
* int SplitScore(int splitIndex, List<GameObject> objects)  
  Tells you how balanced the tree would be if you split objects at position splitIndex. Higher scores are better. (The actual score is just the size of the smaller of the two subtrees you would be making).

# Building the tree

You will start by filling in the method for building the tree:

* BSPTreeNode BuildTree(List<GameObject> objects)

This method is located in the BSPTree class in the file BSPTree.cs.

Again, the basic algorithm is:

BuildTree(objects)  
 if objects.Count = =1  
 return new BSPTreeLeaf(objects[0])  
 Else  
 find the best splitting plane  
 return new BSPTreeInteriorNode(splittingPlane,  
 BuildTree(objects behind plane),  
 BuildTree(objects in front of plane))

The hard part, as we said is finding the splitting plane. Unfortunately, there’s no good way to do this except to try a lot of planes. So we try looking at planes across each of the axes (X, Y, and Z), and then try picking planes at different points along these axes, looking for the best one. So our algorithm for finding the best plane is:

For each axis {  
 Sort objects along axis  
 for i=0 to objects.Count-1 {  
 if splitting here would be a more even split than the current best split {  
 Make a plane pointing along axis, positioned between objects[i-1] and objects[i]  
 if the plane divides the objects   
 this is the new best plane  
 }  
 }  
}

That’s how we find the best plane.

Finally, there’s the question of what we do if we can’t find a splitting plane at all. In a real system, you’d have to give up and choose some plane that divides *most* of the objects, but has a one or more objects that pass through the plane. Then you’d need to modify the tree building algorithm to detect those objects that straddle the plane, and add them to *both* subtrees. However, that would complicate the assignment significantly to little pedagogical advantage. So for this assignment, you need only detect the case where there is no splitting plane and throw a NoSplitAxisException.

Here’s the pseudocode for the whole algorithm:

if objects.Count == 1  
 return new BSPTreeLeaf(objects[0])  
else {

// *Find the best splitting plane*// *keep track of the best plane, the index it splits at, and its score*bestScore = bestIndex = 0  
bestPlane = new Plane()  
  
// *Try each axis. To make it easy to iterate, we give you an array called “axes” with*// *all the axes in it*  
Foreach axis in axes {  
 Sort objects along axis using SortGameObjects  
  
 // *Try each possible pair of objects to split between*  
 for i=0 to objects.Count-1 {  
 // *Is position i a more balanced split than our current best split?*  
 if SplitScore(i, objects)> bestScore {  
 // *Yes. Try splitting here*  
 Make a plane along axis between positions i-1 and i using SplittingPlane  
 if plane SplitsAt position i {  
 // *This is a better splitting plan and index, so remember it*  
 bestPlane = this plane  
 bestIndex = i  
 bestScore = SplitScore(i, objects)  
 }  
 }  
 }  
}

if (bestScore == 0) throw new NoSplitAxisException()  
// *We have the best plane actually split the objects along it, build the subtrees*  
// *for either side, and make a new tree node from them.*  
Resort along axis bestPlane.Normal  
backTree = BuildTree(objects.GetRange(0, bestIndex))  
frontTree = BuildTree(the rest of objects)  
return new BSPTreeInteriorNode(bestPlane, backTree, frontTree)

}

# Walking the tree

The final thing you need to implement is a procedure that, given a BoundingBox, finds all the objects in the BSP tree that intersect that bounding box, and calls a procedure on them:

* virtual void DoAllIntersectingObjects(BoundingBox bbox, TreeFunction function)

When called on BSPTreeLeaf, you can simply take the leaf’s GameObject, ask for its BoundingBox, and then use its Intersects() method to ask it the object’s bounding box overlaps bbox. If so, call function on the object.

When called on a BSPTreeInteriorNode, you need to recursively call the DoAllIntersectingObjects on the node’s front subtree, its back subtree, or both, depending on whether the bounding box is in front of, behind, or intersecting the plane, respectively. Again, you can get this information just by calling the Intersects() method of the plane on the bounding box (or vice-versa), which will return a PlaneIntersectionType (see “New Data Types”, above).

# Trying it out

The tests provided in BSPTreeTests.cs will allow you to test your code against how it is supposed to run. As usual, you are provided with tests that test the smallest amount of functionality up to the complete construction and searching of a BSP tree. There are tests that will assure that you are choosing the correct nodes as well as making sure you’re correctly choosing what is in front and what is behind. Make sure to look at the tests in advance to gain a better understanding of how the code runs and what its structure is.

# Turning it in

As before, to turn the assignment in, you should:

* Choose “Clean solution” from the Build menu in Visual Studio. This will remove binary files and bring your directory down to a manageable size.
* Exit Visual Studio
* Make a ZIP file of your assignment’s directory. Please use ZIP format, not RAR, tgz, or other formats.
* Upload the zip file to blackboard.

1. Given the name, you might expect List<T> to be a linked list, but it’s actually a dynamic array. Java works the same way. [↑](#footnote-ref-1)