Assignment 4
COMP 250, Winter 2022

Prepared by T.A. Kavosh with help from Prof. Mike & TA's Manas and Zoe
Posted: Tues. March 22 , 2022
Due: Wed. April 6 at 23:59 (midnight)

[document last modified: March 31, 2022]

General instructions

- Search for the keyword *updated* to find any places where the PDF has been updated.
- We will provide you with some examples to test your code. See file ExposedTests.java which is part of the starter code. If you pass all these tests, you will get at least 60/100. These tests correspond exactly to the exposed tests on Ed. We will also use private tests as in previous assignments.

  We strongly encourage you to come up with more creative test cases, and to share these test cases on the discussion board. There is a crucial distinction between sharing test cases and sharing solution code. We encourage the former, whereas the latter is a serious academic offense.
- *Ed should be used for code submission only. You should write, compile and test your code locally (e.g. in Eclipse or IntelliJ) for development.*
- For this assignment, we will test more thoroughly how long your code takes to run. Thus, each Ed submission will take longer than for previous assignments. For this reason, you should avoid submitting close to the deadline since it is possible that the server may not be able to handle the load of many last minute submissions.
- **Policy on Academic Integrity:** See the Course Outline Sec. 7 for the general policies. You are also expected to read the posted checklist PDF that specifies what is allowed and what is not allowed on the assignment(s).
- **Late policy:** Late assignments will be accepted up to two days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Please see the submission instructions at the end of this document for more details.

1 Introduction

In this assignment you'll see a practical example of how trees and recursion can be used. We will consider the problem of *collision detection* in computer graphics and animation.

The goal will be to accelerate an interactive 2D physics simulation involving many balls bouncing around like in some chaotic billiards-like game. Like any modern realtime game, our simulation will run in an infinite loop called the "game loop" which in each iteration will update the scene and display a new "frame", or image, to your screen.

For every frame, our physics engine must first detect when two balls are "overlapping" or 'in contact with" or "colliding with" each other, then resolve the contact by separating the two balls and modelling a momentum transfer between them (aka bouncing). It then updates the positions of all the balls using their new velocities. These updates are done in the code provided to you. FYI, you can learn more about these problems in COMP 559 Computer Animation if you are so inclined.

Your task is simply to write the collision detection code for our simulation. You will be given a list of circles and you will need to return a list of all the "contacts" between any two pairs of circles in the list. We want you to do this in two different ways.

1.1 Naive approach

Your first inclination may be to write a nested for loop over all the circles to check *every* pair for potential contact. Indeed this would work, and we'd like for you to implement this, but this is not the optimal approach as its time complexity is $\mathcal{O}(n^2)$ where n is the number of circles. To get a feel for how bad this is, imagine there were 100,000 circles. To check every pair for contact would result in 10 billion (10^{10}) circle-to-circle contact queries which is an absurd amount and would be far too slow.

1.2 Divide-and-conquer approach

If you think a bit you'll realize that checking for contact between circles on opposite ends of the screen is a waste of time, as you can only be in contact with other circles that are "nearby". In other words, you can dramatically speed up the simulation by reducing the "search space" of each circle. We generally perform such search-space reductions by doing a first pass over the data and building a data structure that allows us to speed up subsequent collision queries. These techniques are applied in many areas like real-time graphics (google "raytracing") and machine learning (google "k-d trees").

In this assignment we'll focus on a particular tree-like acceleration structure called a *Bounding Volume Hierarchy* (BVH). As its name suggests, a BVH is a hierarchy or tree consisting of "bounding volumes". A bounding volume is simply a volume, or more generally shape, which bounds or encloses its child volumes. In this assignment we'll use 2D *axis-aligned bounding boxes* as bounding volumes; these are rectangles whose left/right edges are vertical and top/bottom edges are horizontal, i.e., they are not rotated. Our BVH will also be be binary, which means that each node in the tree will have at most two children. We will store the circles within the leaves of this tree.

So you can imagine the root of our BVH as a box with at most two children, where each child is also a box with at most two children, and so on until you reach a leaf node containing one circle. The child boxes of any node in our tree must be nested fully *inside* their parent's box, but sibling boxes are allowed to overlap each other.

The intuitive purpose of this tree is again to reduce the search space for collision detection, i.e., to allow us to quickly *rule out* circles which *cannot* be in contact. You'll soon see the full BVH construction algorithm, but for now you can imagine that we put all the balls on the left side of the screen in a big box, and all the balls on the right side in another box. Now imagine you're a ball and you want to see which other balls you're touching. Suppose you look around and notice that you are not in contact with the right-side box. This means that you cannot be in contact with any of the balls which are *inside* that box, effectively halving the search space. This search-space halving gets recursively repeated at every level of the tree. Similar to a binary search, this allows us to cut the per-ball collision detection complexity to $\mathcal{O}(\log n)$ on average.

Why do we say *on average*? Well, for context, recall the lectures on quicksort. The first step in quicksort is to pick a pivot and partition the input list into two disjoint lists, one with elements which are smaller than the pivot, and one with elements which are larger. Ideally, both of these lists will be of equal length, which leads to a quicksort runtime of $\mathcal{O}(n \log n)$, but if the split is unbalanced then you may encounter quicksort's worst-case runtime of $\mathcal{O}(n^2)$. However, *on average*, if your list is full of random numbers, you can expect the runtime to be "closer" to $\mathcal{O}(n \log n)$.

A very similar phenomena is at play in our BVH. The construction algorithm (explained later) contains a **`split()`** method which takes a list of circles and spatially partitions them. If the split at each node perfectly halves the input list, then the height of our BVH will be $\log n$. In practice, however, our splits may be slightly worse due to the random positioning of the balls, leading to a BVH of height greater than $\log n$. The height of the tree matters because it gives us an upper bound on the number of circle-boundingBox contact queries we must make to reach a leaf in the BVH.

Since the circles are moving, you'll need to build a new BVH tree every frame. This takes time $\mathcal{O}(n \log n)$ in the best case and $\mathcal{O}(n^2)$ in the worst case. This behavior is analogous to quicksort, and we will leave it to you to figure out this analogy.

2 Classes

2.1 `Visualizer` class

You don't need to understand this class for this assignment, but you should run it. Essentially, it runs the main physics simulation and rendering loop in the following order : (1) all contacts are detected using naive or accelerated means; (2) detected contacts are resolved, meaning the balls are assigned new velocities simulating a bounce (3) new positions of all balls are calculated using the velocities and (4) balls at these new positions are displayed.

To play around with the simulation, you can simply use your mouse to grab a ball and drag it around. The balls will then bounce off each other and off the walls. You can also release a ball to "throw it". A demonstration video of the expected behaviour can be found here . If you press the "r" key on your keyboard, the balls will be reset to a random initial configuration.

Circles which are in contact with other circles will be colored blue, and circles which are not in contact with any circles will be colored white. We expect you to use the visualizer in part to debug your assignment and to see what's going on. There are a few fields you should be aware of in this class.

- **`bvhAccelerated`** - If this is false, then the visualizer will use **`getContactsNaive()`** to find all the contacts. If this is true, it will use **`getContactsBVH()`** to find all the contacts and will also draw all the boundingBoxes in your BVH in red. This will show you the structure of your BVH.

- **`simulate`** - If this is set to **`true`**, then circles will bounce off each other and off the walls. If it is set to **`false`**, then circles will not move unless you grab them and drag them around. You may want to set this to false and use the white/blue coloring to debug your contact finders. Once they're all working, you can set it back to **`true`** and have some fun bouncing balls around.

The **`initScene()`** method is responsible for filling the scene with randomly placed balls. To debug a failed contact test, you may want to replace the body of this method by just setting **`circles`** to **PublicTesterUtils.randomInit**() with the appropriate parameters to visualize the scene used for testing. The error messages in the tests will tell you what parameters to use.

2.2 `Box` class

Each **`Box`** object is defined by two corner points **`bottomLeft`** and **`topRight`**. You don't need to modify anything here, but will need to use several methods of this class.

The Box class has several methods that are given in the starter code, including:

- getter methods **`getWidth()`**, **`getHeight()`**, **`getMidX()`**, and **`getMidY()`** that you will use to implement **`BVH.split()`**.
- **`intersectBox()`** which takes a **`Box b`** and returns true if **`this`** and **`b`** are overlapping, and false otherwise. You'll need to use this method in **`getContactBVH()`** described later.

2.3 `Circle` class

Each **`Circle`** object has a **`position`**, **`velocity`**, **`radius`**, and **`id`**. This id is unique to each circle, i.e., no two circles can share the same id. You must use this id to ensure that a circle does not "contact" itself.

Each circle has a **`getBoundingBox()`** method which returns the tightest Box that fully encloses the circle. This is used by the **`BVH.buildtightBoundingBox()`** method described in the next section.

Each circle also has an **`isContacting()`** method, which takes a Circle **c** and returns a **`ContactResult`** object if **this** Circle and **c** are in contact, **and null otherwise**. For our physics simulation to work properly it needs to know slightly more than whether or not Circle **a** and Circle **b** are in contact. It also needs to know the relative direction and distance between the centers of the two circles, which is exactly the information contained in the returned ContactResult. You do not need to modify this method.

2.4 `BVH` class

A bounding volume hierarchy is a rooted tree. In the lectures, we modelled (rooted) trees using two separate classes "tree" and "node", and indeed this is commonly done. In this assignment we take a different approach and define only a single "node" class called **`BVH`**. This is arguably more elegant as every "node" in a tree is just the root of another tree, and so you only *need* a single class. On the other hand, a disadvantage of doing this from an object oriented perspective is that it exposes the underlying data structure: users of the class would have access to the nodes of the tree.

Each BVH object has fields **`boundingBox, child1, child2,`** and **`containedCircle`**. The following method is given to you:

- **`buildTightBoundingBox()`** – a static method which takes an ArrayList of Circle objects and returns the Box that encloses all the circles in the list as tightly as possible.

You must implement the following methods:

- A constructor **`BVH(ArrayList<Circle> circles)`** that first initializes the boundingBox of **this** BVH using the above helper method **`buildTightBoundingBox()`**. Then, if **`circles`** contains more than one element, the method partitions the list into two disjoint lists using **`split()`** – see next – and then uses these two lists to recursively construct and initialize **`this.child1`** and **`this.child2`**. If, however, **`circles`** contains only one element, then it assigns it to **`this.containedCircle`**. This means the **`containedCircle`** field will be non-null only in the leaves of our BVH.

- A static method **`split()`** which takes in an ArrayList **`circles`** of Circle objects and a Box **`boundingBox`** which tightly encloses every circle in **`circles`**, and returns an array of two **`ArrayList<Circle>`** objects. [Updated: March 31. As noted here and here on Ed Discussion, you can create this array using syntax **`new ArrayList[2]`** or **`new ArrayList[]{arraylist1, arraylist2}`**.

  This method is responsible for partitioning **`circles`** into two disjoint ArrayLists of Circle objects, similar to the partitioning phase of quicksort. There actually exist many reasonable ways to do this, and as such we won't enforce a particular splitting strategy so long as the tree you produce is not horribly unbalanced.

  A common splitting strategy which we recommend you implement is to first determine the longest axis of **`boundingBox`** and then partition the circles based on whether the corresponding component of their center point is less than or greater than the midpoint of the boundingBox along that axis.

  As an example, suppose the width of **`boundingBox`** is greater than its height. This means we want to split the box along the **x** axis. We can use one of the methods of the Box class to find the midpoint of the box along the **x** axis. We can then create two ArrayLists of Circle objects from which we define the "left" and "right" children, and then iterate over each Circle object, placing it into the appropriate list based on its **`position.x`** field. The case where height is greater than width is analogous, except we split along the **y** axis instead. We then return the two ArrayLists in an array *i.e.* of length 2.

  IMPORTANT: We deliberately did not specify which child should be assigned to which list, or what do to if there's a tie between the width and height, or what to do if a circle falls *directly* on the splitting plane, or any other trivial detail. This is because the tests *do not care* about the *exact* structure of your tree. Just do whatever you think is reasonable and the exposed tests will tell you if we agree. If you believe your tree is reasonable and it fails the exposed bvh tests, please make a post on the discussion board so we can improve our tests. We also will not be testing what your BVH does when there exist two circles in the same exact position.

- A method **`iterator()`** which returns a new BVHIterator() object – see the **`BVHIterator`** class later.

- Make sure to set the **`Visualizer.bvhAccelerated`** field to true if you want to visualize your BVH or use your BVH contact results in the simulation.

2.5 `ContactFinder` class

This class is responsible for detecting contacts between circles. You will notice that all methods here are static i.e., they only need the arguments passed to them in order to perform their task. Details about a contact are stored in **`ContactResult`** objects (see below).

You must implement the following methods:

- **`getContactsNaive(ArrayList<Circle> circles)`** calculates contacts using the naive $\mathcal{O}(n^2)$ approach described earlier, and returns them as a **`HashSet<ContactResult>`**. You should check if a circle is in contact with every other circle using **`isContacting()`**, but make sure to use the circle's **`id`** so as to not produce any self contacts.

  Note that the ordering of **`a`** and **`b`** in a ContactResult **DOES NOT** matter. This means that we don't care if you do **`a.isContacting(b)`** or **`b.isContacting(a)`** for two circles **`a`** and **`b`** because their return values are equivalent. Because of this, you can add both of them to the HashSet without producing duplicate contacts.

  Also note that a circle may be in contact with *many* other circles simultaneously. So you cannot early terminate once you find the first contact.

- **`getContactsBVH(ArrayList<Circle> circles, BVH bvh)`** uses the BVH acceleration data structure you constructed earlier to find *all* contacts in the scene, and returns them as a **`HashSet<ContactResult>`**. This method accepts a list of circles and a BVH object and so for each circle in the input list, you should call the following method and accumulate any contacts that are found.

  Note that you can use the enhanced for loop to iterate over a **`HashSet`**.

- **`getContactBVH(Circle c, BVH bvh)`** finds contacts between **`c`** and the circles contained in the leaves of **`bvh`**, again returned as a **`HashSet<ContactResult>`**. This is where the main "acceleration" takes place - you quickly discard circles that cannot intersect with **`c`** by checking against their enclosing **`BVHs`**.

  - If the bounding box of **`c`** does not intersect bounding box of **`bvh`**, return an empty hashset of **`ContactResult`**.
  - If **`bvh`** is a leaf node, check if **`c`** is in contact with this leaf node's **`containedCircle`**. If so, add to the HashSet of **`ContactResult`** and return the HashSet.
  - If **`bvh`** is a non-leaf node and it's children are non-null, then call the method recursively on the children. You should add any contacts detected within the children to the HashSet and return it.

  This method should be efficient. In particular you should not traverse the entire tree, since this would defeat the purpose of the tree! Only traverse the nodes whose boundingBoxes you are in contact with.

  You must again use the circle ids to avoid contacts between a circle and itself.

  Note that like in **`getContactsNaive()`**, we don't care about the ordering of the two circles in your contactResults, and that a circle may be in contact with *many* other circles simultaneously so you cannot early terminate once you find the first contact.

2.6 `ContactResult` class

This class holds information about a contact once it has been detected. You don't need to modify this class and only need to understand it at a high level. **`Circle a, b`** are the two circles in contact, **`contactNormal`** is a vector pointing from one circle to the other and **`penetrationDepth`** is the amount of overlap between two circles. The last two variables are used for physics calculations. Note that the "contact" model here allows the circles to interpenetrate slightly before they bounce off each other.

2.7 `BVHIterator` class [Updated Mar.23: inner class of `BVH`]

In order to demonstrate how custom iterators can be created, we would like you to implement an iterator for the BVH datatype. Note that this portion of the assignment is completely unrelated to the above contact-finding portion. As such, you **SHOULD NOT** use your iterator in your contact-finding code.

The iterator we'd like for you to build will allow you to use a for-each loop to iterate over all the circles contained within a BVH as demonstrated in the exposed test **`BVHIteratorTest()`**. To do this, you will first need to implement the following methods inside the **`BVHIterator`** class, which implements the **`Iterator<Circle>`** interface:

- **`BVHIterator(BVH bvh)`** populates a list with all the circles contained within **`bvh`** using some recursive traversal. Calling the **`next()`** method then allows a user of this class to move through the list which you constructed. You may create and initialize any member variables you want to implement this logic, and so are allowed to import data structures provided by Java.
- **`hasNext()`** returns true if there are circles left to be iterated over, else false.
- **`next()`** returns the next circle. We **do not** care about the order in which these are returned.

Finally, notice that there is a method **`iterator()`** in the **`BVH`** class which you must implement. This method should simply returns a new **`BVHIterator()`** object. This method is what enables you to write **`for (Circle c : bvh)`** . **DO NOT FORGET TO IMPLEMENT THIS METHOD.**

2.8 `Tester` class

This class contains exposed tests to help with your implementation. Feel free to look at individual test cases to see what exactly is being tested.

Your Task

Implement the required methods in the **`BVH`**, **`ContactFinder`** and **`BVHIterator`** classes as described above.

Submission

Please follow the instructions on Ed Lessons Assignment 4.

- Ed does not want you to have a package name. Therefore you should remove the package name before uploading to Ed.
- We will check that your file imports only what you need, namely the imports given in the starter code.
- You may submit multiple times. Your assignment will graded using your most recent submission.
- If you submit code that does not compile, you will automatically receive a grade of 0. Since we grade only your latest submission, you must ensure that your latest submission compiles!
- [Updated Mar. 26] The deadline is midnight Wed. April 6. On Ed, this deadline is coded as 12:00 AM on Thurs. Apr. 7. Similarly, on Ed, the two day window for late submission ends at midnight midnight Fri. April 8 but is encoded as 12:00 AM on Sat. April 9.

Good luck and happy coding!